GROUP – 11:

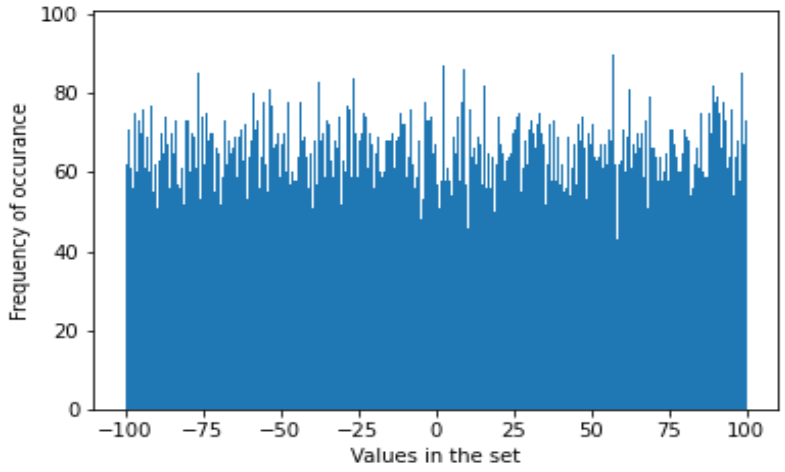
GROUP MEMBERS:

1) **RISHITA DE**
2) **DEBARGHYA DEY**
3) **KINGSHUK BARUA**
4) **DEBASMITA DAS**
5) **SATTWIK BARUA**

1A uniform data set

In [1]:

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

UD = np.random.uniform(-100, 100, 2**16)
#-100 lowest value, 100 highest value
y = np.arange(2**16)

_ = plt.hist(UD, bins = 1000)
_ = plt.xlabel('Values in the set' )
_ = plt.ylabel('Frequency of occurance' )
```

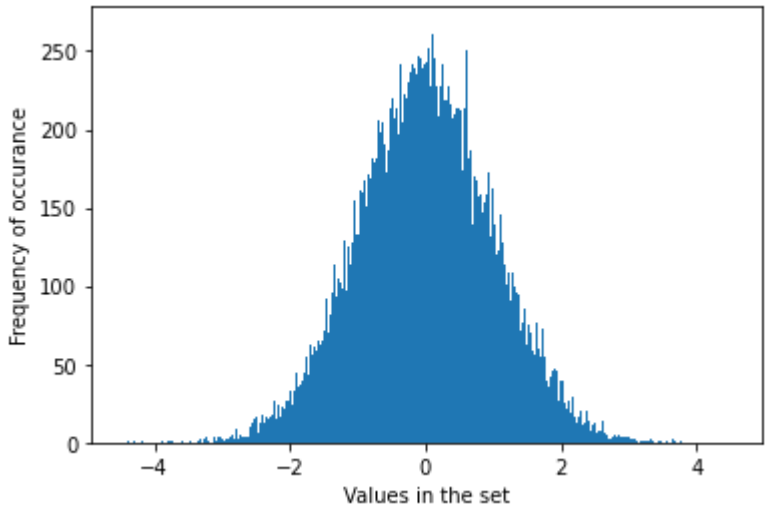


In

In [2]:
```
#_ = plt.plot(UD, y)
```

1B normal distribution data set

[3]:
```
mu, stddev = 0, 1 # mean = 0 and standard deviation = 1
ND = np.random.normal(mu, stddev, 2**16)
_ = plt.hist(ND, bins = 1000)
_ = plt.xlabel('Values in the set')
_ = plt.ylabel('Frequency of occurance')
```



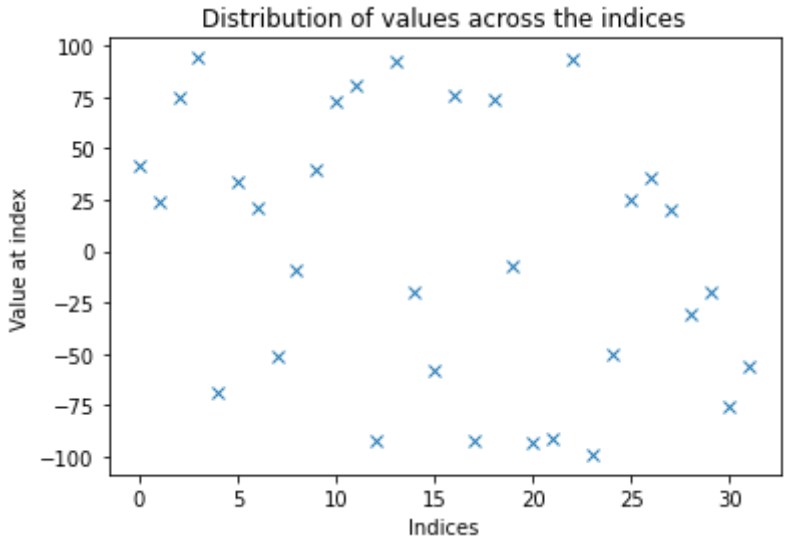
In [4]:
```
#_= plt.plot(ND, y)
```

In [5]:
```
UD = np.ndarray.tolist(UD)
ND = np.ndarray.tolist(ND)
```

In [6]:

visualization of the uniform dataset

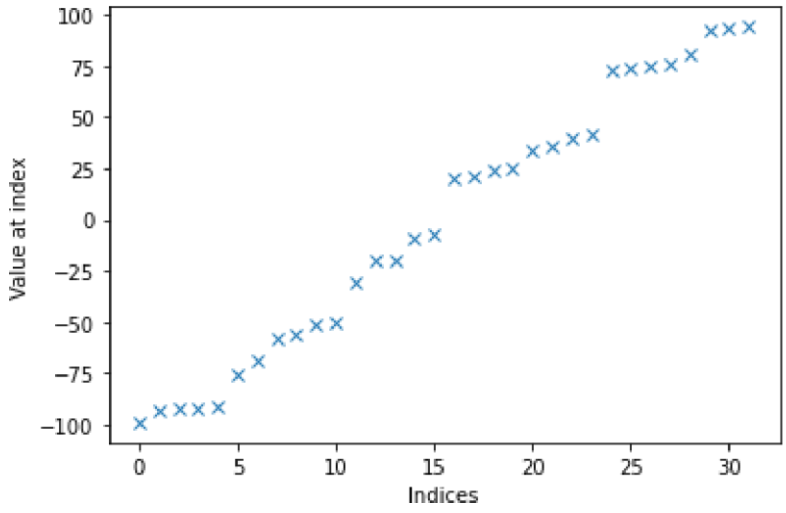
```
set1 = UD[ : 2**5]
xaxis =[ i for i in range(len(set1))]
_ = plt.plot(xaxis, set1, marker = 'x', linestyle = 'none') #plots the values of the data s
_ = plt.xlabel('Indices')
_ = plt.ylabel('Value at index')
_= plt.title('Distribution of values across the indices')
```



2A merge sort on UD

In [7]:

```
set1 = UD[ : 2**5]

def mergeSort(arr):
    if len(arr) > 1:
        mid = len(arr)//2

        L = arr[:mid]
        R = arr[mid:]
        mergeSort(L)
        mergeSort(R)

        i = j = k = 0

        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1

        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1

mergeSort(set1)
xaxis =[ i for i in range(len(set1))]
_ = plt.plot(xaxis, set1, marker = 'x', linestyle = 'none') #plots the values of the data s
#non decreasing graph shows that set is sorted
_ = plt.xlabel('Indices')
_ = plt.ylabel('Value at index')
```



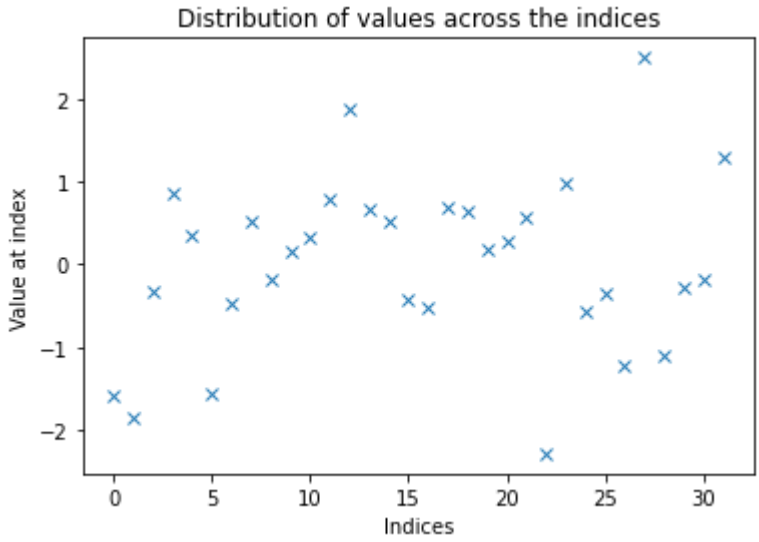
In [8]:

```
#prints the actual values of the sorted array

#for i in range(len(set1)):
#    print("%f\t" %set1[i])
```

visualization of normal data set

In [9]:

```
set2 = ND[ : 2**5]
xaxis =[ i for i in range(len(set2))]
_ = plt.plot(xaxis, set2, marker = 'x', linestyle = 'none') #plots the values of the data s
_ = plt.xlabel('Indices')
_ = plt.ylabel('Value at index')
_= plt.title('Distribution of values across the indices')
```

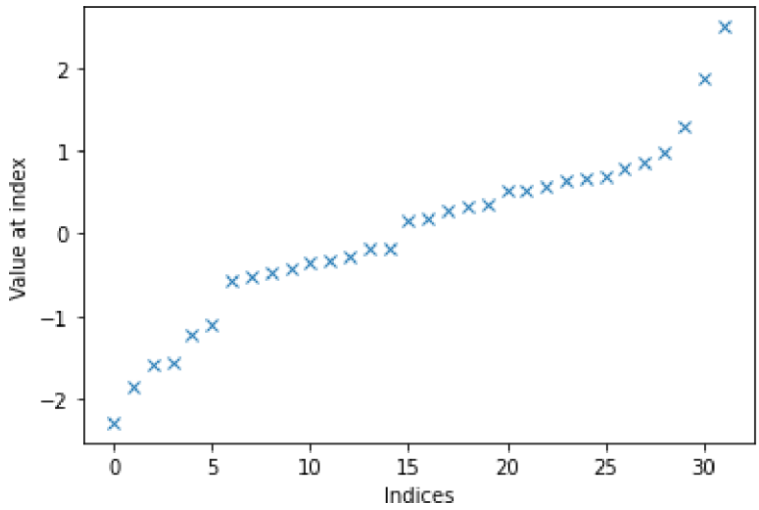


In

2B quick sort on Normal data set

[10]:

```
set2 = ND[ : 2**5]

def partition(arr, low, high):
    i = (low-1)
    pivot = arr[high]
    for j in range(low, high):
        if arr[j] <= pivot:
            i = i+1
            arr[i], arr[j] = arr[j], arr[i]

    arr[i+1], arr[high] = arr[high], arr[i+1]
    return (i+1)

def quickSort(arr, low, high):
    if len(arr) == 1:
        return arr
    if low < high:
        pi = partition(arr, low, high)
        quickSort(arr, low, pi-1)
        quickSort(arr, pi+1, high)

n = len(set2)
quickSort(set2, 0, n-1)
xaxis =[ i for i in range(len(set2))]
_ = plt.plot(xaxis, set2, marker = 'x', linestyle = 'none') #plots the values of the data s
#non decreasing graph shows the sorted order
_ = plt.xlabel('Indices')
_ = plt.ylabel('Value at index')
```



In [10]:

3 Merge sort performance on Uniform distributiondataset

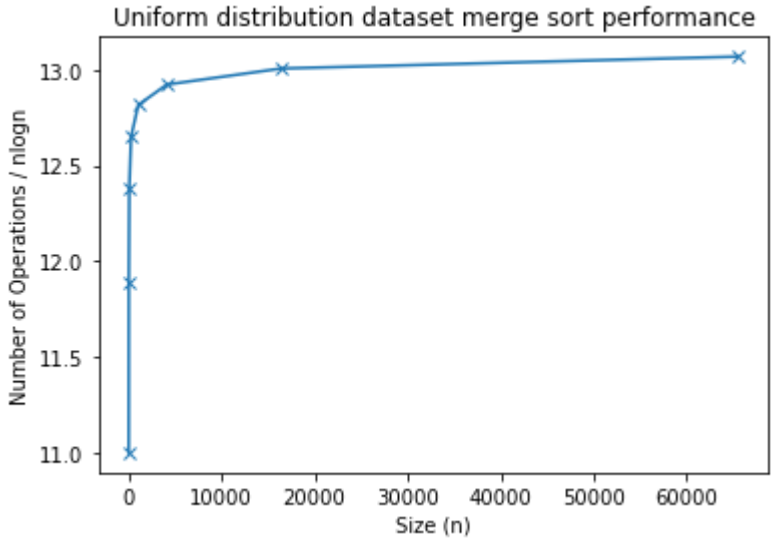
[11]:

```
import math

def mergeSortCount(arr, count):
    total = count
    if len(arr) > 1:
        mid = len(arr)//2
        L = arr[:mid]
        R = arr[mid:]
        count += 4
        total += mergeSortCount(L, count)
        total += mergeSortCount(R, count)

        i = j = k = 0
        total += 3

        while i < len(L) and j < len(R):
            total += 2
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
                total += 3
            else:
                arr[k] = R[j]
                j += 1
                total += 2
            k += 1
            total += 1

        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1
            total += 4

        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1
            total += 4
    return total

opCountUD_MS = []

for i in range(2, 17, 2):
    set1 = UD[ : 2**i]

    countOfOperationsAndComparisons_UD_MS_set1 = mergeSortCount(set1,0 )
    opCountUD_MS.append(countOfOperationsAndComparisons_UD_MS_set1)


axis = [2**i for i in range(2,17,2)]
i = 0
for num, n in zip(opCountUD_MS, axis):
    num = num / (n * math.log(n, 2))
    print(num)
    opCountUD_MS[i] = num
    i += 1

_ = plt.plot(axis, opCountUD_MS, marker = 'x')
```

In

```
_= plt.xlabel('Size (n)')
_ = plt.ylabel('Number of Operations / nlogn')
_ = plt.title('Uniform distribution dataset merge sort performance')
```

```
11.0
11.890625
12.3828125
12.65087890625
12.81650390625
12.923258463541666 13.007167271205358
13.069348335266113
```

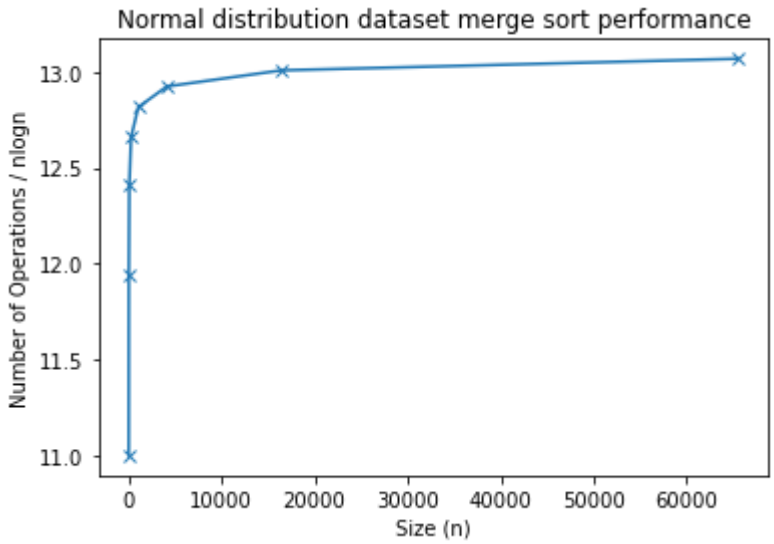


In [12]:

merge sort on normal dataset

```
opCountND_MS = []

for i in range(2, 17, 2):
    set1 = ND[ : 2**i]

    countOfOperationsAndComparisons_ND_MS_set1 = mergeSortCount(set1,0 )
    opCountND_MS.append(countOfOperationsAndComparisons_ND_MS_set1)


axis = [2**i for i in range(2,17,2)]
i = 0
for num, n in zip(opCountND_MS, axis):
    num = num / (n * math.log(n, 2))
    print(num)
    opCountND_MS[i] = num
    i += 1

_ = plt.plot(axis, opCountND_MS, marker = 'x')
_= plt.xlabel('Size (n)')
_ = plt.ylabel('Number of Operations / nlogn')
_ = plt.title('Normal distribution dataset merge sort performance')
```

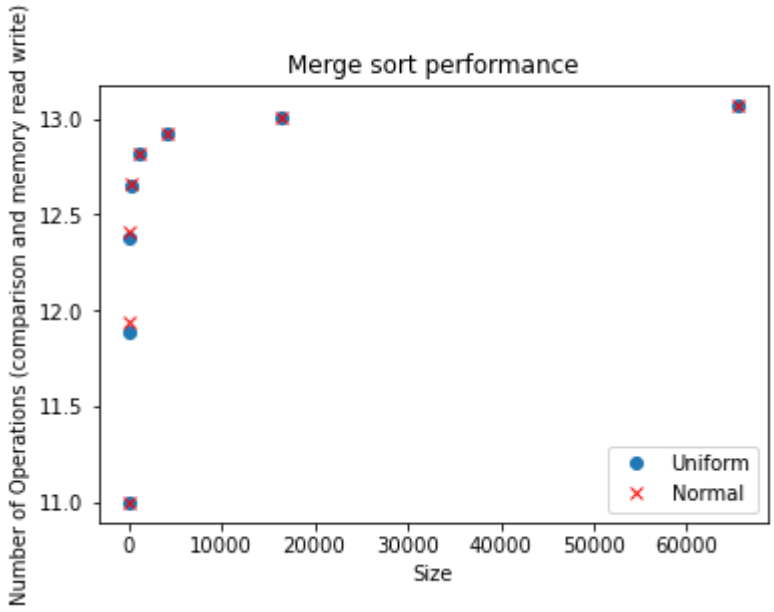
```
11.0
11.9375
12.408854166666666
12.6591796875
12.81572265625
12.92401123046875
13.007829938616071
13.068604469299316
```



merge sort on the datasets

In [13]:

```
_ = plt.plot(axis, opCountUD_MS, marker = 'o', linestyle = 'none', label = 'Uniform')
_ = plt.plot(axis, opCountND_MS, marker = 'x', linestyle = 'none', color = 'red', label = '
_ = plt.ylabel('Number of Operations (comparison and memory read write)')
_ = plt.xlabel('Size')
_ = plt.legend()
_=plt.title('Merge sort performance')
plt.show()

print("SIZE\tUNIFORM\t\tNORMAL")
for size, uniform, normal in zip(axis, opCountUD_MS, opCountND_MS):
    print ("%d\t%f\t\t%f" %(size, uniform, normal))
```



SIZE	UNIFORM	NORMAL
4	11.000000	11.000000
16	11.890625	11.937500
64	12.382812	12.408854
256	12.650879	12.659180
1024	12.816504	12.815723
4096	12.923258	12.924011
16384	13.007167	13.007830
65536	13.069348	13.068604

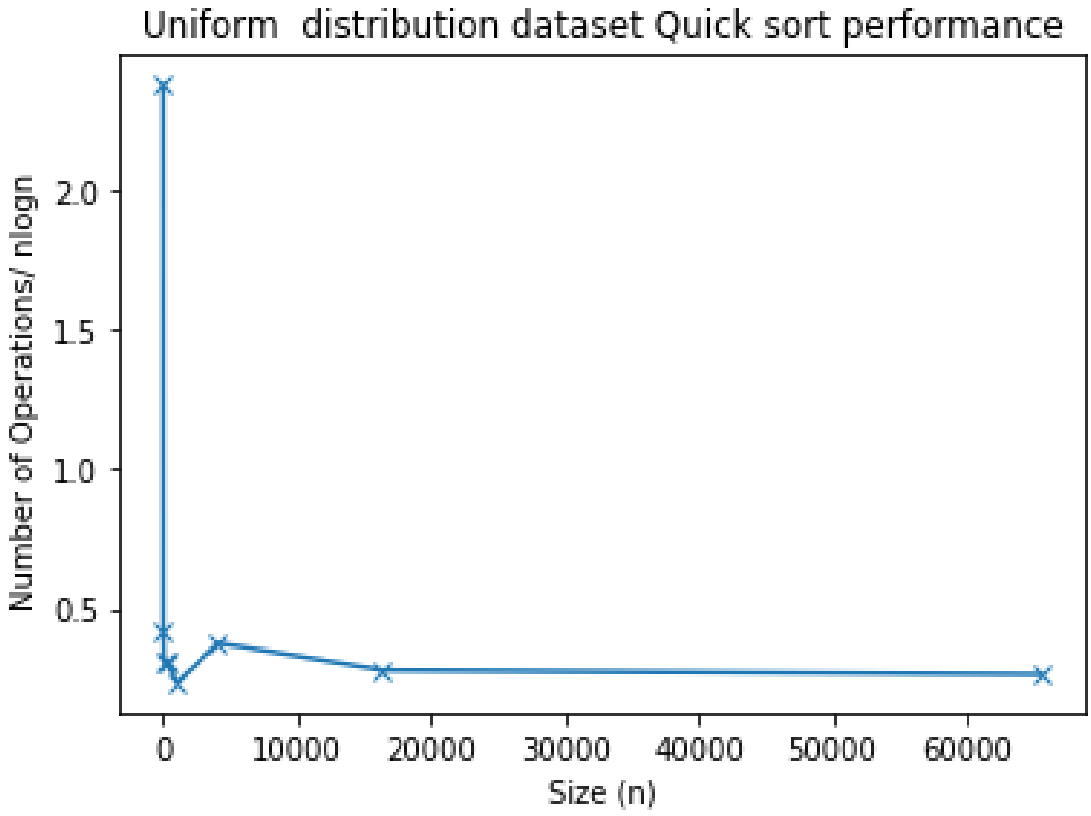
In [14]:

Quicksort performance on uniform distribution dataset

```
def partitionCount(arr, low, high, count):
    i = (low-1)
    pivot = arr[high]
    count += 2

    for j in range(low, high):
        count += 1
        if arr[j] <= pivot:
            i = i+1
            arr[i], arr[j] = arr[j], arr[i]
            count += 4

    arr[i+1], arr[high] = arr[high], arr[i+1]
    count += 2
    return (i+1, count)

def quickSortCount(arr, low, high, total):
    if len(arr) == 1:
        total += 1
        return arr, total
    if low < high:
        pi, count = partitionCount(arr, low, high,total)
        total += count
        quickSortCount(arr, low, pi-1, total)
        quickSortCount(arr, pi+1, high, total)
    return total

opCountUD_QS = []

for i in range(2, 17, 2):
    set2 = UD[ : 2**i]
    n = len(set2)
    opCount = quickSortCount(set2, 0, n-1, 0)
    opCountUD_QS.append(opCount)

axis = [2**i for i in range(2,17,2)]

i = 0
for num, n in zip(opCountUD_QS, axis):
    num = num / (n * math.log(n, 2))
    print(num)
    opCountUD_QS[i] = num
    i += 1

_ = plt.plot(axis, opCountUD_QS, marker = 'x')
_= plt.xlabel('Size (n)')
_ = plt.ylabel('Number of Operations/ nlogn ')
_ = plt.title('Uniform  distribution dataset Quick sort performance')
```

```
2.375
```

```
0.421875
0.3098958333333333
0.30810546875
0.23623046875
0.37750244140625
0.282012939453125 0.2676572799682617
```



In [16]:

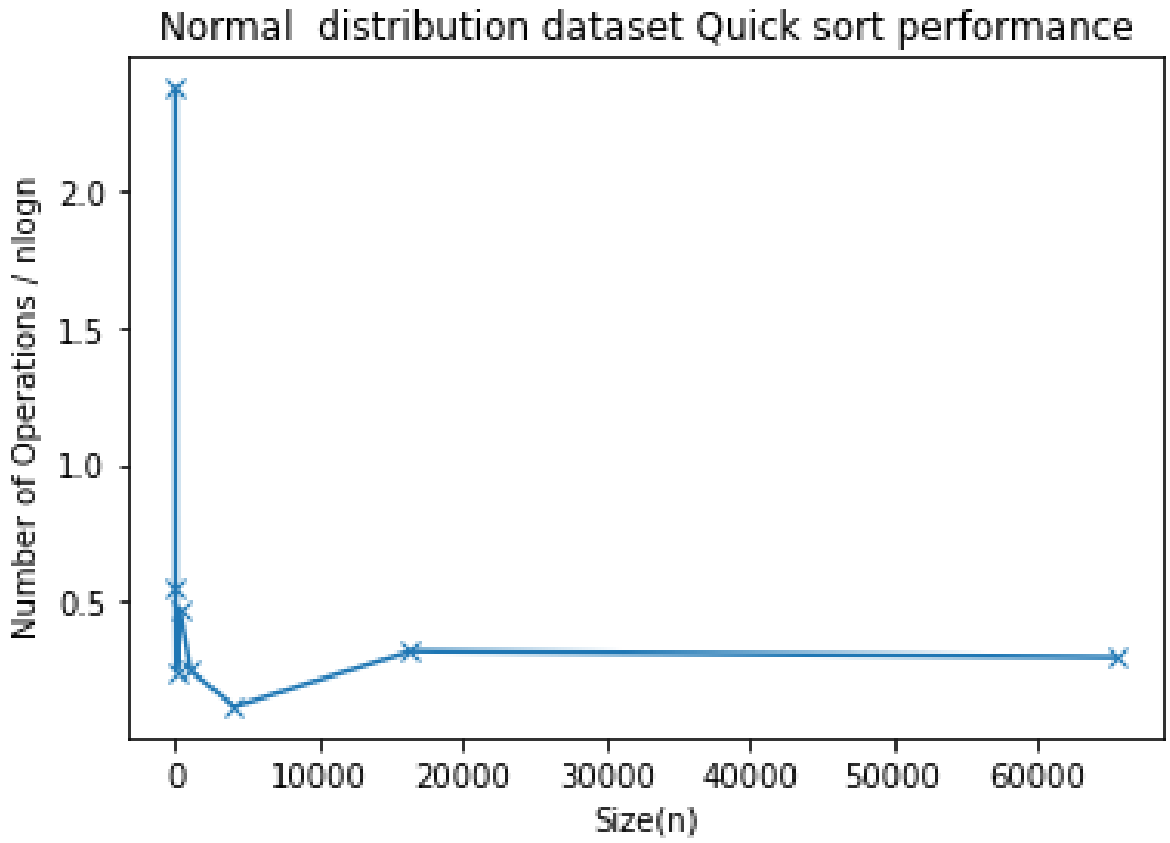
Quicksort on normal

```
opCountND_QS = []

for i in range(2, 17, 2):
    set2 = ND[ : 2**i]
    n = len(set2)
    opCount = quickSortCount(set2, 0, n-1, 0)
    opCountND_QS.append(opCount)

print(opCountND_QS)

axis = [2**i for i in range(2,17,2)]

i = 0
for num, n in zip(opCountND_QS, axis):
    num = num / (n * math.log(n, 2))
    print(num)
    opCountND_QS[i] = num
    i += 1

_ = plt.plot(axis, opCountND_QS, marker = 'x')
_= plt.xlabel('Size(n)')
_ = plt.ylabel('Number of Operations / nlogn ')
_ = plt.title('Normal  distribution dataset Quick sort performance')
```

```
[19, 35, 95, 975, 2555, 5763, 73367, 312431]
```

In [15]:

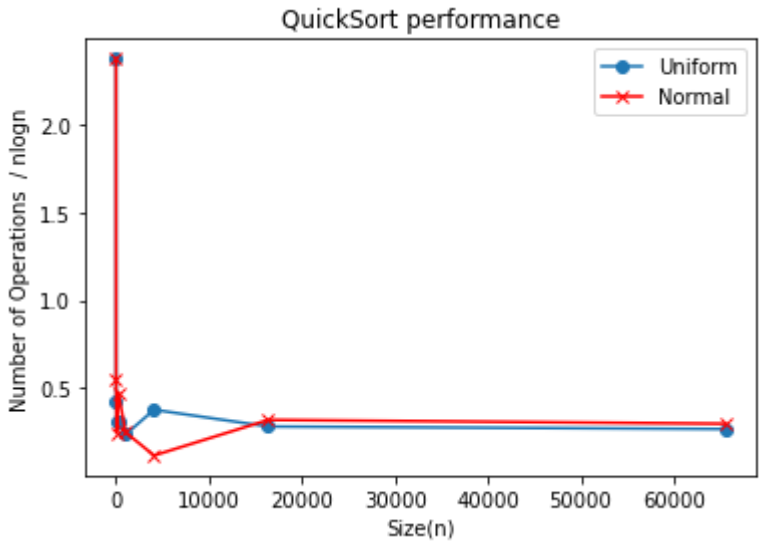
```
2.375
0.546875
0.24739583333333334
0.47607421875
0.24951171875
0.11724853515625
0.319854736328125
0.2979574203491211
```



In [16]:

Quick sort performance

```
_ = plt.plot(axis, opCountUD_QS, marker = 'o', label = 'Uniform')
_ = plt.plot(axis, opCountND_QS, marker = 'x', color = 'red', label = 'Normal')
_ = plt.ylabel('Number of Operations  / nlogn')
_ = plt.xlabel('Size(n)')
_ = plt.legend()
_=plt.title('QuickSort performance')
plt.show()

print("SIZE\tUNIFORM\tNORMAL")
for size, uniform, normal in zip(axis, opCountUD_QS, opCountND_QS):
    print ("%d\t%f\t%f" %(size, uniform, normal))
```




```
SIZE    UNIFORM NORMAL
4       2.375000        2.375000
16      0.421875        0.546875
64      0.309896        0.247396
256     0.308105        0.476074
1024    0.236230        0.249512
4096    0.377502        0.117249
16384 0.282013          0.319855
65536 0.267657          0.297957
```

In [17]:

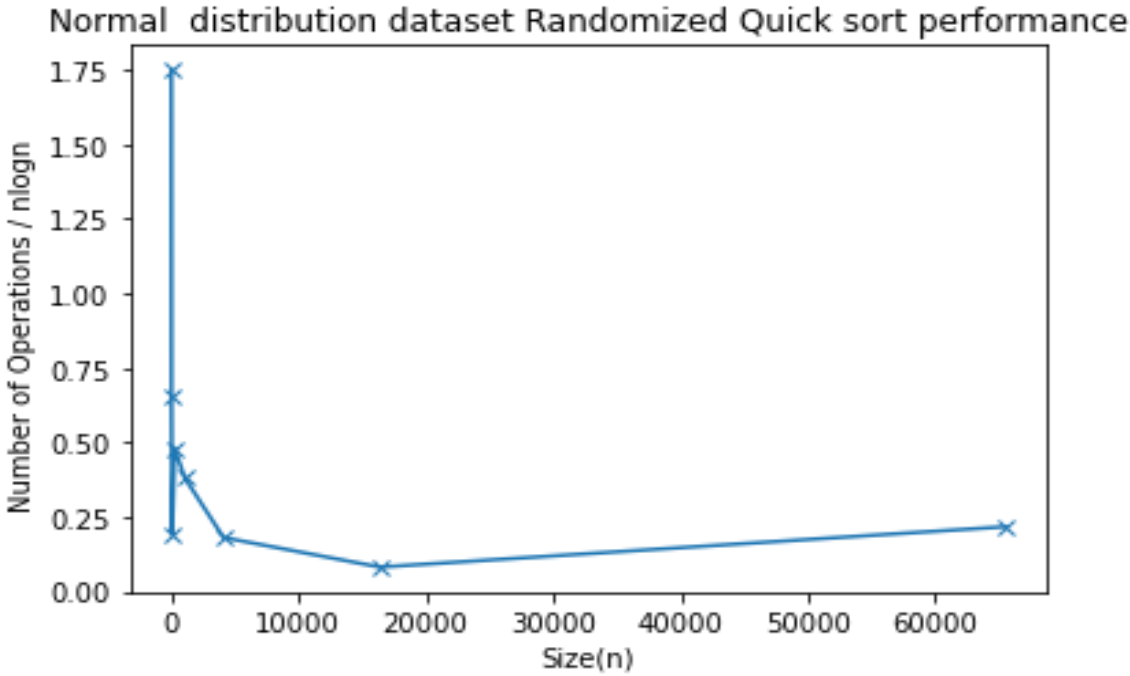
4 Randomized Quick sort

```
import random

def quickSort(lst, a, b, count):
    total = count
    if a < b:
        pivot, total = partition(lst, a, b, total)
        total += 1
        quickSort(lst, a, pivot-1, total)
        quickSort(lst, pivot+1, b, total)
    return total



def partition(lst, a ,b, total):
    random_index = random.randint(a,b)
    lst[b], lst[random_index] = lst[random_index], lst[b]

    pivot = a
    total += 4

    for i in range(a,b):
        total += 1
        if lst[i] < lst[b]:
            lst[i],lst[pivot] = lst[pivot],lst[i]
            pivot += 1
            total += 4
    lst[pivot],lst[b] = lst[b],lst[pivot]
    total += 2
    return pivot, total

opCountND_RQS = []

for i in range(2, 17, 2):
    set2 = ND[ : 2**i]
    n = len(set2)
    opCount = quickSort(set2, 0, n-1, 0)
    opCountND_RQS.append(opCount)

axis = [2**i for i in range(2,17,2)]

i = 0
for num, n in zip(opCountND_RQS, axis):
    num = num / (n * math.log(n, 2))
    print(num)
    opCountND_RQS[i] = num
    i += 1

_ = plt.plot(axis, opCountND_RQS, marker = 'x')
_= plt.xlabel('Size(n)')
_ = plt.ylabel('Number of Operations / nlogn ')
_ = plt.title('Normal  distribution dataset Randomized Quick sort performance')
```

```
1.75
0.65625
```

In [18]:

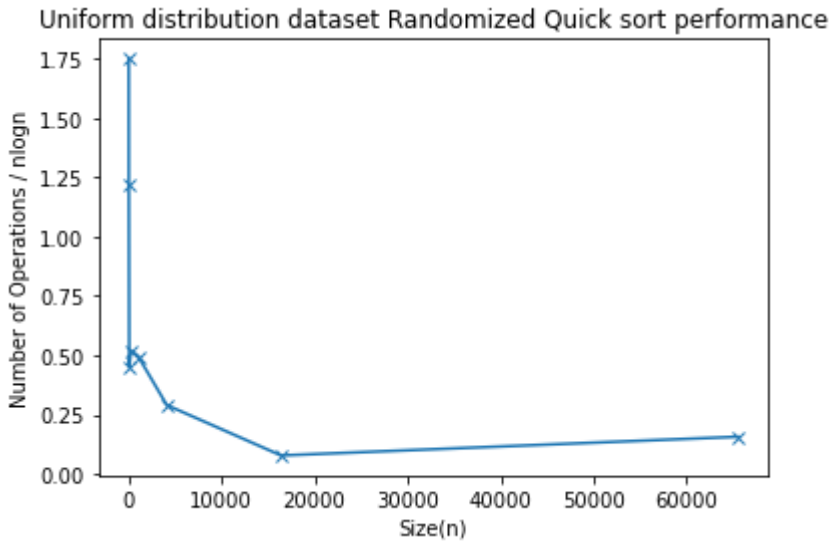
```
0.19270833333333334
0.4814453125
0.3830078125
0.1805419921875
0.08137730189732142
0.2169780731201172
```



In [18]:

```
opCountUD_RQS = []

for i in range(2, 17, 2):
    set2 = UD[ : 2**i]
    n = len(set2)
    opCount = quickSort(set2, 0, n-1, 0)
    opCountUD_RQS.append(opCount)

axis = [2**i for i in range(2,17,2)]

i = 0
for num, n in zip(opCountUD_RQS, axis):
    num = num / (n * math.log(n, 2))
    print(num)
    opCountUD_RQS[i] = num
    i += 1


_ = plt.plot(axis, opCountUD_RQS, marker = 'x')
_= plt.xlabel('Size(n)')
_ = plt.ylabel('Number of Operations / nlogn')
_ = plt.title('Uniform distribution dataset Randomized Quick sort performance')
```

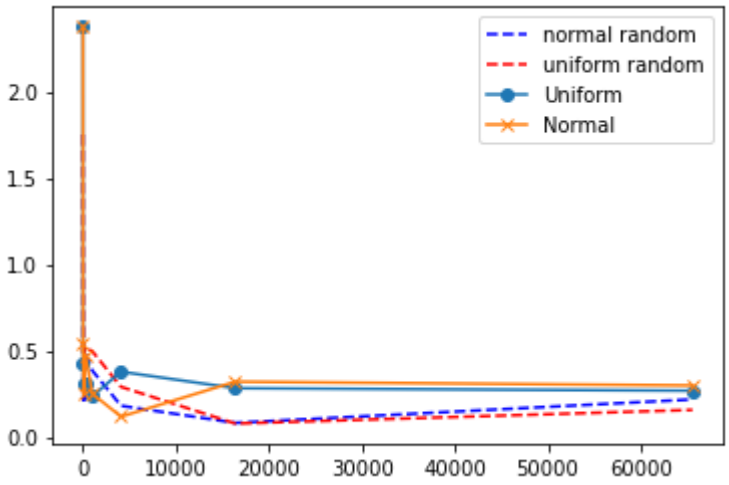
```
 1.75
1.21875
0.453125
0.5185546875
0.4951171875
0.2899983723958333
0.07750592912946429
0.15645408630371094
```



randomized quick sort performance

In [23]:

```
axis = [2**i for i in range(2,17,2)]
_ = plt.plot(axis, opCountND_RQS, '--b' , label = 'normal random')
_ = plt.plot(axis, opCountUD_RQS, '--r', label = 'uniform random')
_ = plt.plot(axis, opCountUD_QS, marker = 'o', label = 'Uniform')
_ = plt.plot(axis, opCountND_QS, marker = 'x' ,label = 'Normal')
_= plt.legend()
plt.show()
print("SIZE\tUNIFORM Random\tUNIFORM\t\tNORMAL Random\tNormal")
for size, uniformr, uniform, normalr, normal in zip(axis, opCountUD_RQS, opCountUD_QS, opCo
    print ("%d\t%f\t%f\t%f\t%f" %(size, uniformr,uniform, normalr, normal))
```




```
SIZE     UNIFORM Random UNIFORM          NORMAL Random Normal
4       1.750000        2.375000        1.750000        2.375000
16      1.218750        0.421875        0.656250        0.546875
64      0.453125        0.309896        0.192708        0.247396
256     0.518555        0.308105        0.481445        0.476074
1024    0.495117        0.236230        0.383008        0.249512
4096    0.289998        0.377502        0.180542        0.117249
16384 0.077506          0.282013        0.081377        0.319855
65536 0.156454          0.267657        0.216978        0.297957
```

In

5) BUCKET SORT:

UNIFORM DISTRIBUTION:

```
#include<iostream>
#include<cstdlib>
#include<ctime>
using namespace std;
struct node
{
    float data;
    struct node *next;
    struct node *prev;
};
class Linked_list
{
    private:
        struct node *head, *tail;
        void sortedInsert(struct node** start, struct node* newNode)
        {
            struct node *ptr;
            if(*start == NULL)
            {
                *start = newNode;
            }
            else if(((*start)->data) >= newNode->data)
            {
                newNode->next = *start;
                newNode->next->prev = newNode;
                *start = newNode;
            }
            else
            {
                ptr = *start;
                while(ptr->next != NULL && ptr->next->data < newNode->data)
                ptr = ptr->next;

                newNode->next = ptr->next;
                if(ptr->next != NULL)
                    newNode->next->prev = newNode;

                ptr->next = newNode;
                newNode->prev = ptr;

            }
        }
    public:
        Linked_list()
        {
            head = NULL;
            tail = NULL;
        }
        void addNode(float n)
        {
            struct node *newnode = new node;
            newnode->data = n;
            newnode->next = NULL;
```

```
            newnode->data = n;
            newnode->next = NULL;
            if(head == NULL)
            {
                newnode->prev = NULL;
                head = newnode;
                tail = newnode;
            }
            else
            {
                tail->next = newnode;
                newnode->prev = tail;
                tail = tail->next;
                tail->prev = newnode->prev;
            }
        }
        void sortLL()
        {
            struct node *sorted = NULL;
            struct node *ptr = head;
            while(ptr != NULL)
            {
                struct node *next = ptr->next;
                ptr->next = ptr->prev = NULL;
                sortedInsert(&sorted, ptr);
                ptr = next;
            }
            head = sorted;
        }
        void concat(float a[], int *pos)
        {
            struct node *ptr;
            ptr = head;
            while(ptr != NULL)
            {
                a[*pos] = ptr->data;
                *pos = *pos + 1;
                ptr = ptr->next;
            }
        }
        void print()
        {
            struct node *ptr;
            ptr = head;
            while(ptr != NULL)
            {
                cout << ptr->data << " ";
                ptr = ptr->next;
            }
        }
};
void bkt_sort(float a[], int size)
{
    Linked_list *b;
    b = new Linked_list[size];
    int i, c, pos = 0;
    for(i = 0; i < size; i++)
    {
        c = a[i]*size;
        b[c].addNode(a[i]);
    }
    for (i = 0; i < size; i++)
    {
        printf("Bucket[%d]: ", i);
        b[i].print();
        cout << endl;
    }
```

In

```
120         for(i = 0; i < size; i++)
121         {
122             b[i].sortLL();
123         }
124         cout << "_____" << endl;
125         cout << "Buckets after sorting: " << endl;
126         for (i = 0; i < size; i++)
127         {
128             printf("Bucket[%d]: ", i);
129             b[i].print();
130             cout << endl;
131         }
132 
133         for(i = 0; i < size; i++)
134         {
135             b[i].concat(a, &pos);
136         }
137     }
138     void dataSet(float a[], int n)
139     {
140         int i;
141         for(i = 0; i < n; i++)
142         {
143             a[i] = (float)(rand())/(float)(RAND_MAX);
144         }
145     }
```

```
146     void printarr(float a[], int size)
147     {
148         int i;
149         for(i = 0; i < size; i++)
150         {
151             cout << a[i] << " ";
152         }
153         cout << endl;
154     }
155     int main()
156     {
157         int n = 10;
158         float *arr;
159         float t;
160         arr = new float[n];
161         srand(time(NULL));
162         dataSet(arr, n);
163         cout << "Original dataset: " << endl;
164         printarr(arr, n);
165         cout << "_____" << endl;
166         bkt_sort(arr, n);
167         cout << "_____" << endl;
168         cout << "Sorted array: " << endl;
169         printarr(arr, n);
170     }
```

OUTPUT :

```
Original dataset:
0.37904 0.616779 0.677084 0.319681 0.989105 0.485672 0.0922575 0.650807 0.337901 0.195898

_____________________________________
Bucket[0]: 0.0922575
Bucket[1]: 0.195898
Bucket[2]:
Bucket[3]: 0.37904 0.319681 0.337901
Bucket[4]: 0.485672
Bucket[5]:
Bucket[6]: 0.616779 0.677084 0.650807
Bucket[7]:
Bucket[8]:
Bucket[9]: 0.989105
_____________________
Buckets after sorting:
Bucket[0]: 0.0922575
Bucket[1]: 0.195898
Bucket[2]:
Bucket[3]: 0.319681 0.337901 0.37904
Bucket[4]: 0.485672
Bucket[5]:
Bucket[6]: 0.616779 0.650807 0.677084
Bucket[7]:
Bucket[8]:
Bucket[9]: 0.989105

_____________________________________
Sorted array:
0.0922575 0.195898 0.319681 0.337901 0.37904 0.485672 0.616779 0.650807 0.677084 0.989105

--------------------------------
```

In

NORMAL DISTRIBUTION:

```
#define SIGMA 1.0
#define MEAN 0.0
#include<iostream>
#include<cstdlib>
#include<ctime>
#include<cmath>
using namespace std;
struct node
{
    float data;
    struct node *next;
    struct node *prev;
};
class Linked_list
{
    private:
        struct node *head, *tail;
        void sortedInsert(struct node** start, struct node* newNode)
        {
            struct node *ptr;
            if(*start == NULL)
            {
                *start = newNode;
            }
```

```
            }
            else if(((*start)->data) >= newNode->data)
            {
                newNode->next = *start;
                newNode->next->prev = newNode;
                *start = newNode;
            }
            else
            {
                ptr = *start;
                while(ptr->next != NULL && ptr->next->data < newNode->data)
                ptr = ptr->next;

                newNode->next = ptr->next;
                if(ptr->next != NULL)
                    newNode->next->prev = newNode;

                ptr->next = newNode;
                newNode->prev = ptr;

            }
        }
    }
public:
    Linked_list()
    {
        head = NULL;
        tail = NULL;
    }
    void addNode(float n)
    {
        struct node *newnode = new node;
        newnode->data = n;
        newnode->next = NULL;
        if(head == NULL)
        {
            newnode->prev = NULL;
            head = newnode;
            tail = newnode;
        }
        else
        {
            tail->next = newnode;
            newnode->prev = tail;
            tail = tail->next;
            tail->prev = newnode->prev;
        }
    }

```

In

```
72          void sortLL()
73          {
74              struct node *sorted = NULL;
75              struct node *ptr = head;
76              while(ptr != NULL)
77              {
78                  struct node *next = ptr->next;
79                  ptr->next = ptr->prev = NULL;
80                  sortedInsert(&sorted, ptr);
81                  ptr = next;
82              }
83              head = sorted;
84          }
85          void concat(float a[], int *pos)
86          {
87              struct node *ptr;
88              ptr = head;
89              while(ptr != NULL)
90              {
91                  a[*pos] = ptr->data;
92                  *pos = *pos + 1;
93                  ptr = ptr->next;
94              }
95          }
```

```
96          void print()
97          {
98              struct node *ptr;
99              ptr = head;
100             while(ptr != NULL)
101             {
102                 cout << ptr->data << " ";
103                 ptr = ptr->next;
104             }
105         }
106 };
107 void bkt_sort(float a[], int size)
108 {
109     Linked_list *b;
110     b = new Linked_list[size];
111     int i, c, pos = 0;
112     for(i = 0; i < size; i++)
113     {
114         c = a[i]*size;
115         b[c].addNode(a[i]);
116     }
117     for (i = 0; i < size; i++)
118     {
119         printf("Bucket[%d]: ", i);
120         b[i].print();
121         cout << endl;
122     }
```

```
123         for(i = 0; i < size; i++)
124         {
125             b[i].sortLL();
126         }
127         cout << "_____" << endl;
128         cout << "Buckets after sorting: " << endl;
129         for (i = 0; i < size; i++)
130         {
131             printf("Bucket[%d]: ", i);
132             b[i].print();
133             cout << endl;
134         }
135
136         for(i = 0; i < size; i++)
137         {
138             b[i].concat(a, &pos);
139         }
140     }
141     void printarr(float a[], int size)
142     {
143         int i;
144         for(i = 0; i < size; i++)
145         {
146             cout << a[i] << " ";
147         }
148         cout << endl;
149     }
```

```
150     float rand_gen()
151     {
152         return ((float)(rand())/((float) RAND_MAX));
153     }
154     float normal_rand()
155     {
156         float y1, y2;
157         y1 = rand_gen();
158         y2 = rand_gen();
159         return (float)(sqrt(-2.*log(y1)) * cos(2*3.14*y2) * 1.0);
160     }
161     void dataSet(float a[], int n)
162     {
163         int i;
164         float no;
165         for(i = 0; i < n; i++)
166         {
167             no = normal_rand();
168             no = no * SIGMA + MEAN;
169             if(no > 1 || no < 0)
170             {
171                 i--;
172                 continue;
173             }
174             a[i] = no;
175         }
176     }
```

In

```
            no = normal_rand();
            no = no * SIGMA + MEAN;
            if(no > 1 || no < 0)
            {
                i--;
                continue;
            }
            a[i] = no;
        }
    }
    int main()
    {
        int n = 10;
        float *arr;
        float t;
        arr = new float[n];
        srand(time(NULL));
        dataSet(arr, n);
        cout << "Original dataset: " << endl;
        printarr(arr, n);
        cout << "_____" << endl;
        bkt_sort(arr, n);
        cout << "_____" << endl;
        cout << "Sorted array: " << endl;
        printarr(arr, n);
    }
```

OUTPUT :

```
Original dataset:
0.924213 0.0241982 0.0235668 0.397037 0.71683 0.206932 0.280131 0.427998 0.961181 0.623286
_____
Bucket[0]: 0.0241982 0.0235668
Bucket[1]:
Bucket[2]: 0.206932 0.280131
Bucket[3]: 0.397037
Bucket[4]: 0.427998
Bucket[5]:
Bucket[6]: 0.623286
Bucket[7]: 0.71683
Bucket[8]:
Bucket[9]: 0.924213 0.961181
_____
Buckets after sorting:
Bucket[0]: 0.0235668 0.0241982
Bucket[1]:
Bucket[2]: 0.206932 0.280131
Bucket[3]: 0.397037
Bucket[4]: 0.427998
Bucket[5]:
Bucket[6]: 0.623286
Bucket[7]: 0.71683
Bucket[8]:
Bucket[9]: 0.924213 0.961181
_____
Sorted array:
0.0235668 0.0241982 0.206932 0.280131 0.397037 0.427998 0.623286 0.71683 0.924213 0.961181
```

AVERAGE TIME COMPLEXITY COMPARISON:

UNIFORM DISTRIBUTION:

```cpp
#include<iostream>
#include<cstdlib>
#include<ctime>
using namespace std;
struct node
{
    float data;
    struct node *next;
    struct node *prev;
};
class Linked_list
{
    private:
        struct node *head, *tail;
        void sortedInsert(struct node** start, struct node* newNode)
        {
            struct node *ptr;
            if(*start == NULL)
            {
                *start = newNode;
            }
            else if(((*start)->data) >= newNode->data)
            {
                newNode->next = *start;
                newNode->next->prev = newNode;
                *start = newNode;
            }
            else
            {
                ptr = *start;
                while(ptr->next != NULL && ptr->next->data < newNode->data)
                ptr = ptr->next;
```

In

```
                newNode->next = ptr->next;
                if(ptr->next != NULL)
                    newNode->next->prev = newNode;

                ptr->next = newNode;
                newNode->prev = ptr;

            }
        }
    public:
        Linked_list()
        {
            head = NULL;
            tail = NULL;
        }
        void addNode(float n)
        {
            struct node *newnode = new node;
            newnode->data = n;
            newnode->next = NULL;
            if(head == NULL)
            {
                newnode->prev = NULL;
                head = newnode;
                tail = newnode;
            }
            else
            {
                tail->next = newnode;
                newnode->prev = tail;
                tail = tail->next;
                tail->prev = newnode->prev;
            }
        }
```

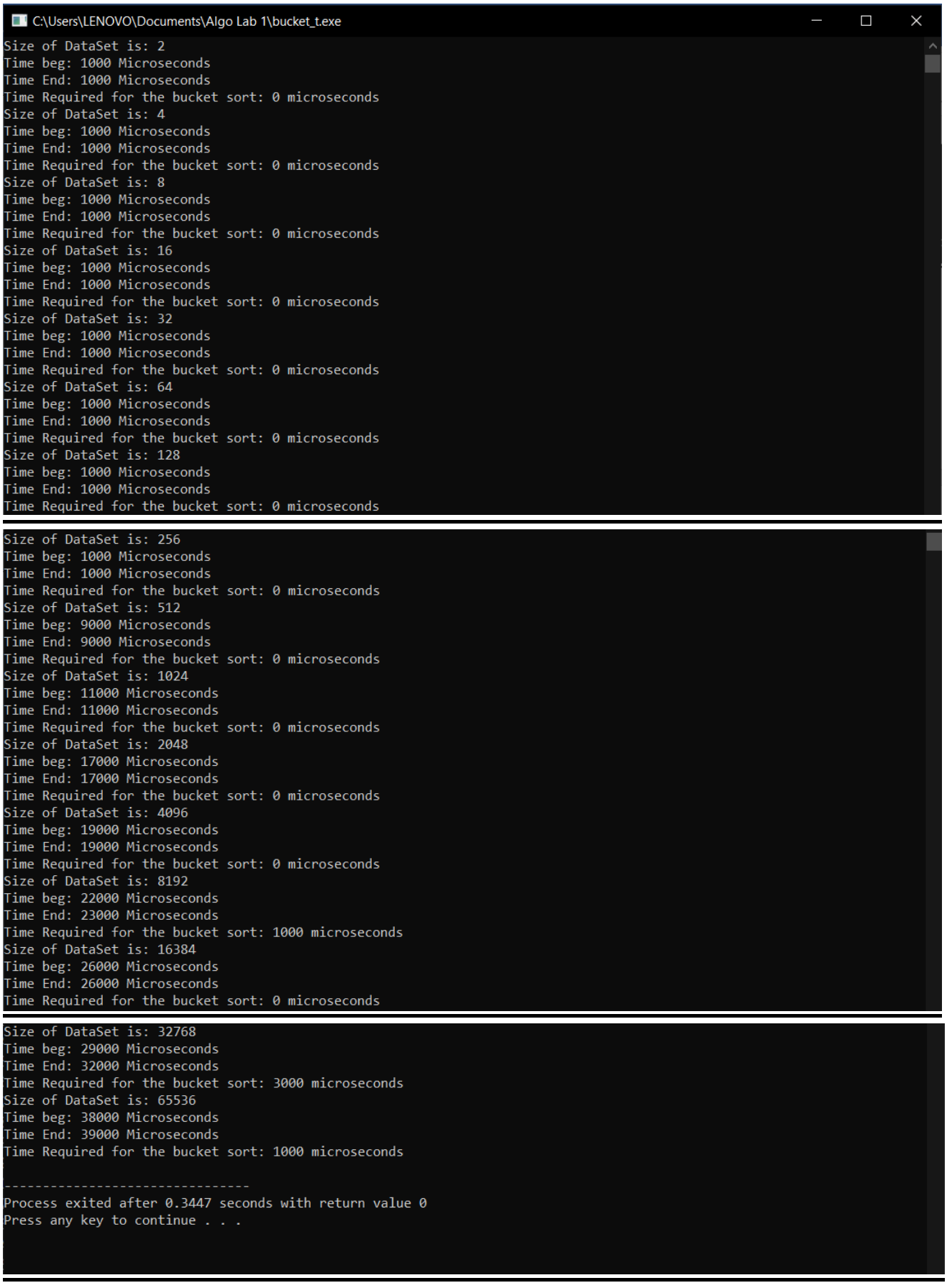
```
        void sortLL()
        {
            struct node *sorted = NULL;
            struct node *ptr = head;
            while(ptr != NULL)
            {
                struct node *next = ptr->next;
                ptr->next = ptr->prev = NULL;
                sortedInsert(&sorted, ptr);
                ptr = next;
            }
            head = sorted;
        }
        void concat(float a[], int *pos)
        {
            struct node *ptr;
            ptr = head;
            while(ptr != NULL)
            {
                a[*pos] = ptr->data;
                *pos = *pos + 1;
                ptr = ptr->next;
            }
        }
};
```

```
94  void bkt_sort(float a[], int size)
95  {
96      Linked_list *b;
97      b = new Linked_list[size];
98      int i, c, pos = 0;
99      for(i = 0; i < size; i++)
100     {
101         c = a[i]*size;
102         b[c].addNode(a[i]);
103     }
104
105     for(i = 0; i < size; i++)
106     {
107         b[i].sortLL();
108     }
109     for(i = 0; i < size; i++)
110     {
111         b[i].concat(a, &pos);
112     }
113 }
114 void dataSet(float a[], int n)
115 {
116     int i;
117     for(i = 0; i < n; i++)
118     {
119         a[i] = (float)(rand())/(float)(RAND_MAX);
120     }
121 }
```

```
112     }
113 }
114 void dataSet(float a[], int n)
115 {
116     int i;
117     for(i = 0; i < n; i++)
118     {
119         a[i] = (float)(rand())/(float)(RAND_MAX);
120     }
121 }
122 int main()
123 {
124     int n = 2, i = 0;
125     float *arr;
126     for(i = 1; i <= 16; i++)
127     {
128         cout << "Size of DataSet is: " << n << endl;
129         arr = new float[n];
130         clock_t time_beg, time_end, time_req;
131         float t;
132         srand(time(0));
133         dataSet(arr, n);
134         time_beg = clock();
135         bkt_sort(arr, n);
136         time_end = clock();
137         time_req = (time_end - time_beg);
138         t = (float) (time_req/(float)CLOCKS_PER_SEC);
139         t = (t * 1000000.0);
140         cout << "Time beg: " << (float)(time_beg/(float)CLOCKS_PER_SEC)*1000000.0 << " Microseconds" << endl;
141         cout << "Time End: " << (float)(time_end/(float)CLOCKS_PER_SEC)*1000000.0 << " Microseconds" << endl;
142
143         cout << "Time Required for the bucket sort: " << t << " microseconds" << endl;
144         n = n*2;
145     }
146 }
```

In

OUTPUT :


C:\Users\LENOVO\Documents\Algo Lab 1\bucket_t.exe
Size of DataSet is: 2
Time beg: 1000 Microseconds
Time End: 1000 Microseconds
Time Required for the bucket sort: 0 microseconds
Size of DataSet is: 4
Time beg: 1000 Microseconds
Time End: 1000 Microseconds
Time Required for the bucket sort: 0 microseconds
Size of DataSet is: 8
Time beg: 1000 Microseconds
Time End: 1000 Microseconds
Time Required for the bucket sort: 0 microseconds
Size of DataSet is: 16
Time beg: 1000 Microseconds
Time End: 1000 Microseconds
Time Required for the bucket sort: 0 microseconds
Size of DataSet is: 32
Time beg: 1000 Microseconds
Time End: 1000 Microseconds
Time Required for the bucket sort: 0 microseconds
Size of DataSet is: 64
Time beg: 1000 Microseconds
Time End: 1000 Microseconds
Time Required for the bucket sort: 0 microseconds
Size of DataSet is: 128
Time beg: 1000 Microseconds
Time End: 1000 Microseconds
Time Required for the bucket sort: 0 microseconds
Size of DataSet is: 256
Time beg: 1000 Microseconds
Time End: 1000 Microseconds
Time Required for the bucket sort: 0 microseconds
Size of DataSet is: 512
Time beg: 9000 Microseconds
Time End: 9000 Microseconds
Time Required for the bucket sort: 0 microseconds
Size of DataSet is: 1024
Time beg: 11000 Microseconds
Time End: 11000 Microseconds
Time Required for the bucket sort: 0 microseconds
Size of DataSet is: 2048
Time beg: 17000 Microseconds
Time End: 17000 Microseconds
Time Required for the bucket sort: 0 microseconds
Size of DataSet is: 4096
Time beg: 19000 Microseconds
Time End: 19000 Microseconds
Time Required for the bucket sort: 0 microseconds
Size of DataSet is: 8192
Time beg: 22000 Microseconds
Time End: 23000 Microseconds
Time Required for the bucket sort: 1000 microseconds
Size of DataSet is: 16384
Time beg: 26000 Microseconds
Time End: 26000 Microseconds
Time Required for the bucket sort: 0 microseconds
Size of DataSet is: 32768
Time beg: 29000 Microseconds
Time End: 32000 Microseconds
Time Required for the bucket sort: 3000 microseconds
Size of DataSet is: 65536
Time beg: 38000 Microseconds
Time End: 39000 Microseconds
Time Required for the bucket sort: 1000 microseconds

Process exited after 0.3447 seconds with return value 0
Press any key to continue . . .

NORMAL DISTRIBUTION:

```
#define SIGMA 1.0
#define MEAN 0.0
#include<iostream>
#include<cstdlib>
#include<ctime>
#include<cmath>
using namespace std;
struct node
{
    float data;
    struct node *next;
    struct node *prev;
};
class Linked_list
{
    private:
        struct node *head, *tail;
        void sortedInsert(struct node** start, struct node* newNode)
        {
            struct node *ptr;
            if(*start == NULL)
            {
                *start = newNode;
            }
            else if(((*start)->data) >= newNode->data)
            {
                newNode->next = *start;
                newNode->next->prev = newNode;
                *start = newNode;
            }
            else
            {
                ptr = *start;
                while(ptr->next != NULL && ptr->next->data < newNode->data)
                ptr = ptr->next;

                newNode->next = ptr->next;
                if(ptr->next != NULL)
                    newNode->next->prev = newNode;

                ptr->next = newNode;
                newNode->prev = ptr;

            }
        }
    public:
        Linked_list()
        {
            head = NULL;
            tail = NULL;
        }
        void addNode(float n)
        {
            struct node *newnode = new node;
            newnode->data = n;
            newnode->next = NULL;
            if(head == NULL)
            {
                newnode->prev = NULL;
                head = newnode;
                tail = newnode;
            }
            else
            {
                tail->next = newnode;
                newnode->prev = tail;
                tail = tail->next;
                tail->prev = newnode->prev;
            }
        }

```

In

```
72          void sortLL()
73          {
74              struct node *sorted = NULL;
75              struct node *ptr = head;
76              while(ptr != NULL)
77              {
78                  struct node *next = ptr->next;
79                  ptr->next = ptr->prev = NULL;
80                  sortedInsert(&sorted, ptr);
81                  ptr = next;
82              }
83              head = sorted;
84          }
85          void concat(float a[], int *pos)
86          {
87              struct node *ptr;
88              ptr = head;
89              while(ptr != NULL)
90              {
91                  a[*pos] = ptr->data;
92                  *pos = *pos + 1;
93                  ptr = ptr->next;
94              }
95          }
96  };
97  void bkt_sort(float a[], int size)
98  {
99      Linked_list *b;
100     b = new Linked_list[size];
101     int i, c, pos = 0;
```

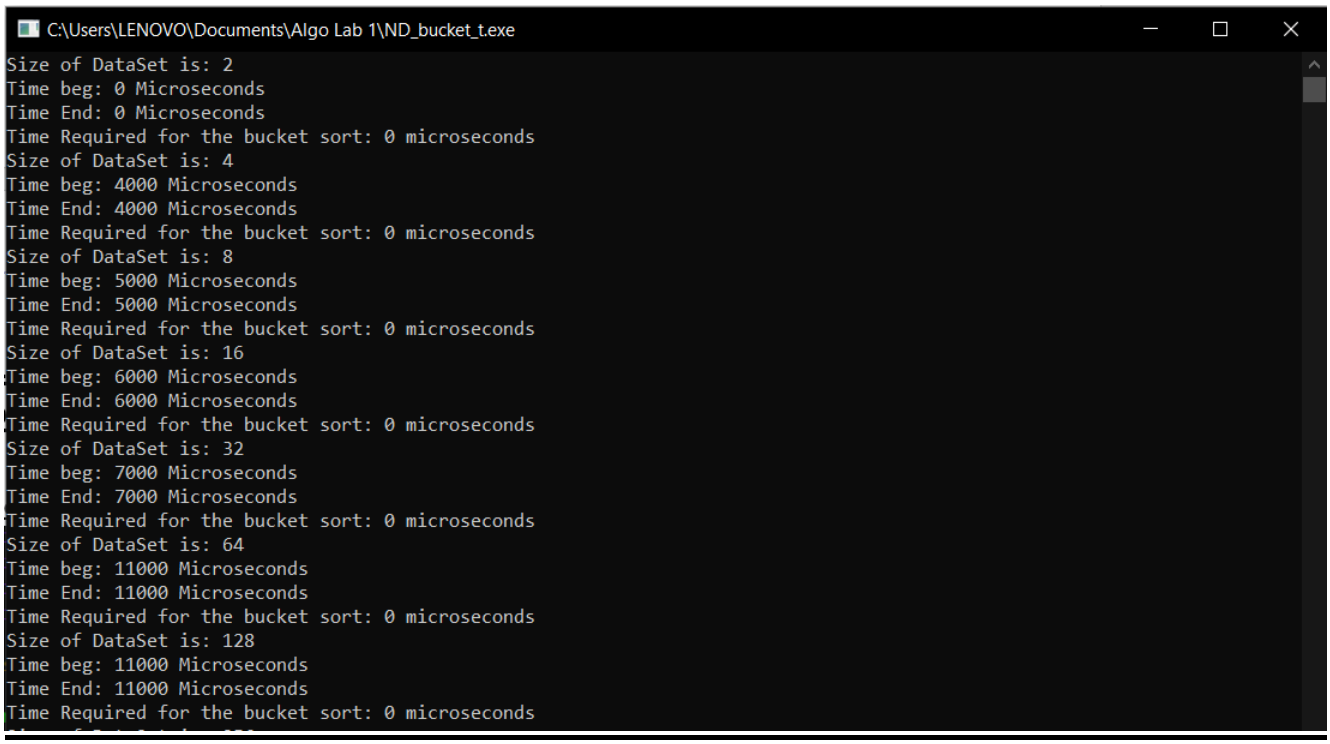
```
102     for(i = 0; i < size; i++)
103     {
104         c = a[i]*size;
105         b[c].addNode(a[i]);
106     }
107
108     for(i = 0; i < size; i++)
109     {
110         b[i].sortLL();
111     }
112
113     for(i = 0; i < size; i++)
114     {
115         b[i].concat(a, &pos);
116     }
117 }
118 void printarr(float a[], int size)
119 {
120     int i;
121     for(i = 0; i < size; i++)
122     {
123         cout << a[i] << " ";
124     }
125     cout << endl;
126 }
127 float rand_gen()
128 {
129     return ((float)(rand())/((float) RAND_MAX));
130 }
```

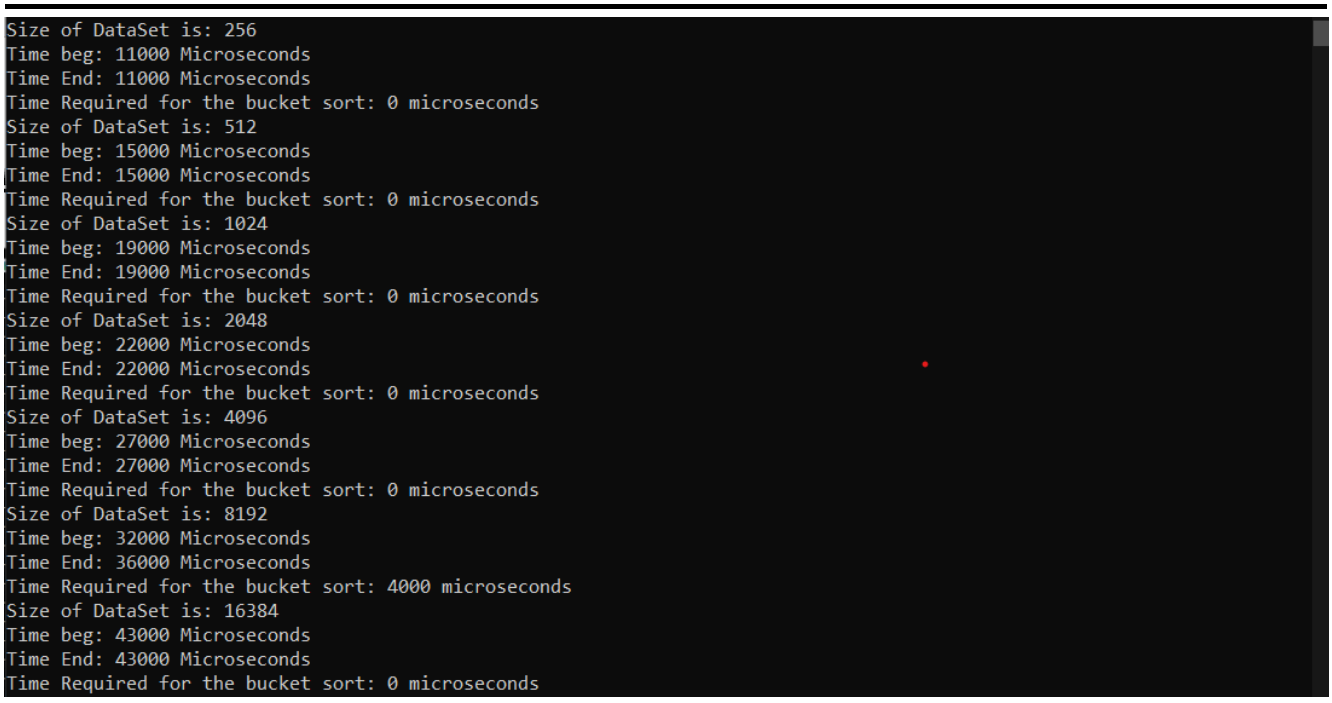
```
131  float normal_rand()
132  {
133      float y1, y2;
134      y1 = rand_gen();
135      y2 = rand_gen();
136      return (float)(sqrt(-2.*log(y1)) * cos(2*3.14*y2) * 1.0);
137  }
138  void dataSet(float a[], int n)
139  {
140      int i;
141      float no;
142      for(i = 0; i < n; i++)
143      {
144          no = normal_rand();
145          no = no * SIGMA + MEAN;
146          if(no > 1 || no < 0)
147          {
148              i--;
149              continue;
150          }
151          a[i] = no;
152      }
153  }
```

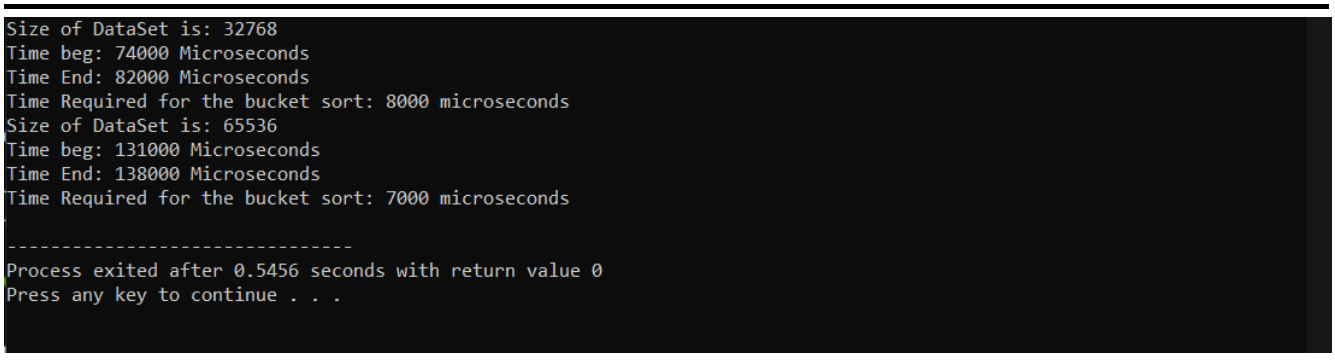
```
154  int main()
155  {
156      int n = 2, i = 0;
157      float *arr;
158      for(i = 1; i <= 16; i++)
159      {
160          cout << "Size of DataSet is: " << n << endl;
161          arr = new float[n];
162          clock_t time_beg, time_end, time_req;
163          float t;
164          srand(time(0));
165          dataSet(arr, n);
166          time_beg = clock();
167          bkt_sort(arr, n);
168          time_end = clock();
169          time_req = (time_end - time_beg);
170          t = (float) (time_req/(float)CLOCKS_PER_SEC);
171          t = (t * 1000000.0);
172          cout << "Time beg: " << (float)(time_beg/(float)CLOCKS_PER_SEC)*1000000.0 << " Microseconds" << endl;
173          cout << "Time End: " << (float)(time_end/(float)CLOCKS_PER_SEC)*1000000.0 << " Microseconds" << endl;
174
175          cout << "Time Required for the bucket sort: " << t << " microseconds" << endl;
176          n = n*2;
177      }
178  }
```

In

OUTPUT :


C:\Users\LENOVO\Documents\Algo Lab 1\ND_bucket_t.exe
Size of DataSet is: 2
Time beg: 0 Microseconds
Time End: 0 Microseconds
Time Required for the bucket sort: 0 microseconds
Size of DataSet is: 4
Time beg: 4000 Microseconds
Time End: 4000 Microseconds
Time Required for the bucket sort: 0 microseconds
Size of DataSet is: 8
Time beg: 5000 Microseconds
Time End: 5000 Microseconds
Time Required for the bucket sort: 0 microseconds
Size of DataSet is: 16
Time beg: 6000 Microseconds
Time End: 6000 Microseconds
Time Required for the bucket sort: 0 microseconds
Size of DataSet is: 32
Time beg: 7000 Microseconds
Time End: 7000 Microseconds
Time Required for the bucket sort: 0 microseconds
Size of DataSet is: 64
Time beg: 11000 Microseconds
Time End: 11000 Microseconds
Time Required for the bucket sort: 0 microseconds
Size of DataSet is: 128
Time beg: 11000 Microseconds
Time End: 11000 Microseconds
Time Required for the bucket sort: 0 microseconds


Size of DataSet is: 256
Time beg: 11000 Microseconds
Time End: 11000 Microseconds
Time Required for the bucket sort: 0 microseconds
Size of DataSet is: 512
Time beg: 15000 Microseconds
Time End: 15000 Microseconds
Time Required for the bucket sort: 0 microseconds
Size of DataSet is: 1024
Time beg: 19000 Microseconds
Time End: 19000 Microseconds
Time Required for the bucket sort: 0 microseconds
Size of DataSet is: 2048
Time beg: 22000 Microseconds
Time End: 22000 Microseconds
Time Required for the bucket sort: 0 microseconds
Size of DataSet is: 4096
Time beg: 27000 Microseconds
Time End: 27000 Microseconds
Time Required for the bucket sort: 0 microseconds
Size of DataSet is: 8192
Time beg: 32000 Microseconds
Time End: 36000 Microseconds
Time Required for the bucket sort: 4000 microseconds
Size of DataSet is: 16384
Time beg: 43000 Microseconds
Time End: 43000 Microseconds
Time Required for the bucket sort: 0 microseconds


Size of DataSet is: 32768
Time beg: 74000 Microseconds
Time End: 82000 Microseconds
Time Required for the bucket sort: 8000 microseconds
Size of DataSet is: 65536
Time beg: 131000 Microseconds
Time End: 138000 Microseconds
Time Required for the bucket sort: 7000 microseconds

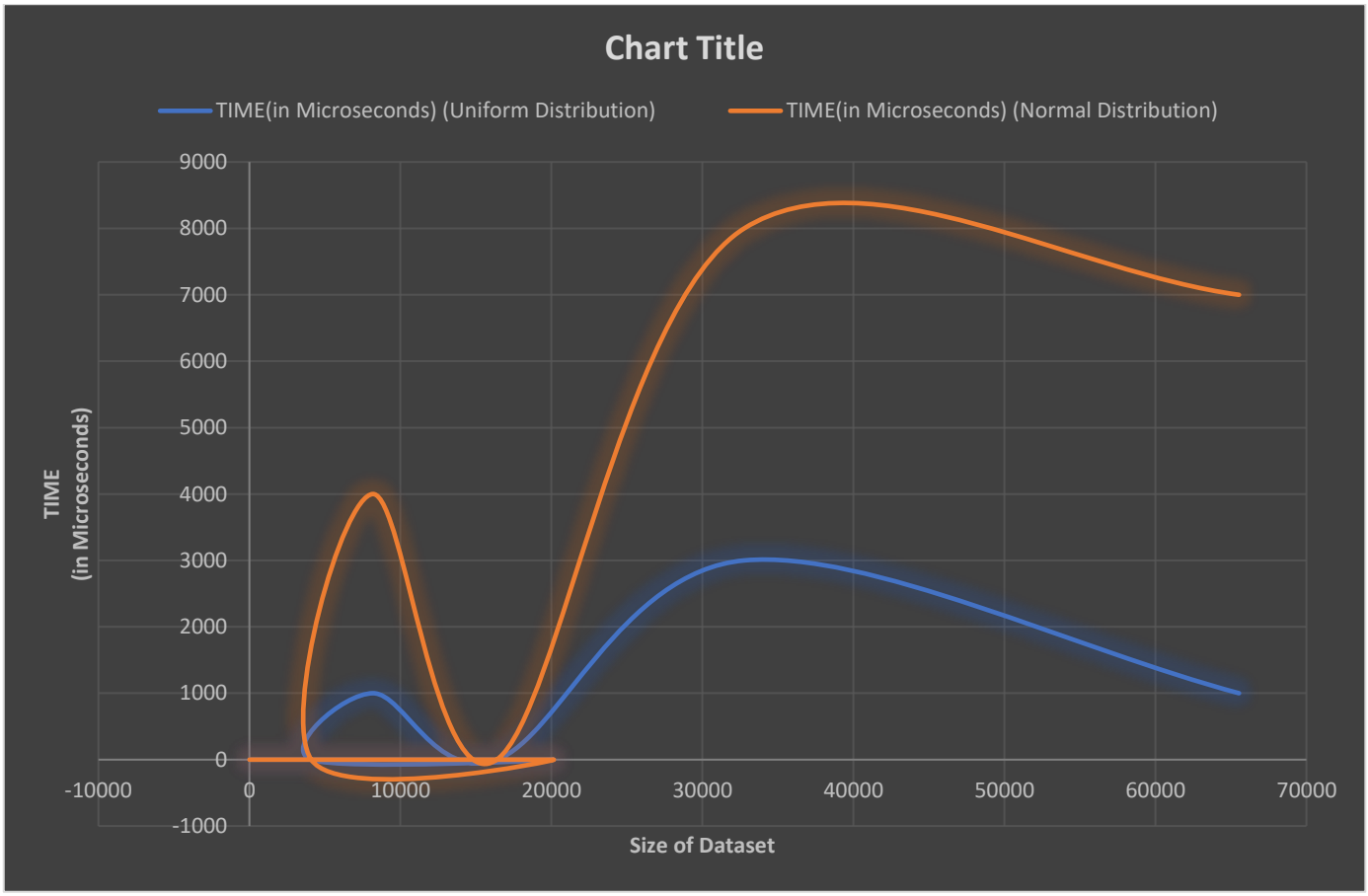
Process exited after 0.5456 seconds with return value 0
Press any key to continue . . .

TABLE :

1	***SIZE OF DATASET***	***TIME(in Microseconds) (Uniform Distribution)***	***TIME(in Microseconds) (Normal Distribution)***
2	2	0	0
3	4	0	0
4	8	0	0
5	16	0	0
6	32	0	0
7	64	0	0
8	128	0	0
9	256	0	0
10	512	0	0
11	1024	0	0
12	20148	0	0
13	4096	0	0
14	8192	1000	4000
15	16384	0	0
16	32768	3000	8000
17	65536	1000	7000

In

CURVE :



INFERENCE :

By looking at the curve(above), we can observe that the TIME curve of Normal Distribution (Orange line) is steeper (has more slope) as compared to the TIME curve of Uniform Distribution (Blue Line). Hence, we can conclude that the Average-Case Time Complexity of the Bucket Sort performed in randomly generated set of elements in Normal Distribution is much higher than that of the Bucket Sort performed in randomly generated set of elements in Uniform Distribution.

Linear median selection algorithm (by taking median of medians) :-

Code in C++ :

```cpp
#include<iostream>
#include<algorithm>
#include<climits>
#include"ud.c"

using namespace std;

int partition(int arr[], int l, int r, int k);

// A simple function to find median of arr[].  This is called
// only for an array of odd size  in this program.
int findMedian(int arr[], int n)
{
    sort(arr, arr+n);  // Sort the array

    return arr[n/2];   // Return middle element
}

//  ASSUMPTION: ALL ELEMENTS IN ARR[] ARE DISTINCT
int kthSmallest(int arr[], int l, int r, int k,int size)
{
    // If k is smaller than number of elements in array
    if (k > 0 && k <= r - l + 1)
    {
        int n = r-l+1; // Number of elements in arr[l..r]

        // Divide arr[] in groups of odd size , calculate median
        // of every group and store it in median[] array.

         int median_array_size = (n% size) == 0 ? (n% size): (n% size)+1;
        int i, median[median_array_size];
        for (i=0; i<n/size; i++)
            median[i] = findMedian(arr+l+i*size, size);
        if (i*size < n) //For last group with less than "size" elements
        {
            median[i] = findMedian(arr+l+i*size, n%size);
            i++;
        }

        // Find median of all medians using recursive call.
        // If median[] has only one element, then no need
        // of recursive call
        int medOfMed = (i == 1)? median[i-1]:
                                 kthSmallest(median, 0, i-1, i/2,size);
```

In

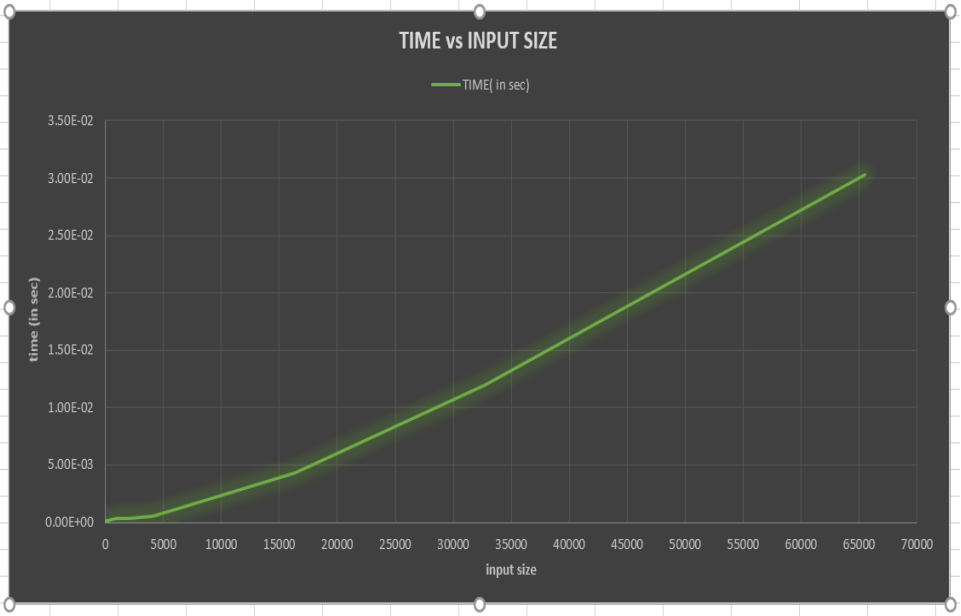
```
        // Partition the array around a random element and
        // get position of pivot element in sorted array
        int pos = partition(arr, l, r, medOfMed);

        // If position is same as k
        if (pos-l == k-1)
            return arr[pos];
        if (pos-l > k-1)  // If position is more, recur for left
            return kthSmallest(arr, l, pos-1, k,size);

        // Else recur for right subarray
        return kthSmallest(arr, pos+1, r, k-pos+l-1,size);
    }

    // If k is more than number of elements in array
    return INT_MAX;
}

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

// It searches for x in arr[l..r], and partitions the array
// around x.
int partition(int arr[], int l, int r, int x)
{
    // Search for x in arr[l..r] and move it to end
    int i;
    for (i=l; i<r; i++)
        if (arr[i] == x)
           break;
    swap(&arr[i], &arr[r]);

    // Standard partition algorithm
    i = l;
    for (int j = l; j <= r - 1; j++)
    {
        if (arr[j] <= x)
        {
            swap(&arr[i], &arr[j]);
            i++;
        }
    }
    swap(&arr[i], &arr[r]);
    return i;
}
```

```
// Driver program to test above methods
int main()
{

    int n ,k,size;
    cout << "Enter the no.of elements and the value of k for which smallest element is to be found\n";
    cin >> n >> k;
    cout << "Enter the size of partition like 3/5/7...\n";
    cin >> size;
    int* arr = new int[n];
    create_ud(arr,n);
    cout << "The original dataset is :\n";
    for(int i = 0; i< n; i++)
       cout << arr[i] << '\n';
    clock_t begin = clock();
    cout << "K'th smallest element is : "
         << kthSmallest(arr, 0, n-1, k,size) << '\n';
    clock_t end = clock();
    double duration = double(end-begin)/CLOCKS_PER_SEC;
    cout << "Time take " << duration;
    delete []arr;
    return 0;
}
```

"ud.c" is the file which contains the code for creating a uniform dataset, and hence helps in calling the function create_ud(arr,n). "nd.c" and create_nd(arr,n) was used for the normal dataset.

Graph :

INPUT	TIME(in sec)
2	8.00E-06
4	1.90E-05
8	2.20E-05
16	8.70E-05
32	1.01E-04
64	1.03E-04
128	1.17E-04
256	1.47E-04
512	2.29E-04
1024	2.98E-04
2048	3.80E-04
4096	5.80E-04
8192	1.80E-03
16384	4.30E-03
32768	1.20E-02
65536	3.03E-02



The graph plotted is line, thus proving the linear time complexity.

Analysis:

The median-of-medians divides a list into sublists of length five to get an optimal running time. Remember, finding the median of small lists by brute force (sorting) takes a small amount of time, so the length of the sublists must be fairly small. However, adjusting the sublist size to three, for example, does change the running time for the worse.

If the algorithm divided the list into sublists of length three, it would cause a worst case "2n/3" recursions, yielding the recurrence T(n) = T(n/3) + T(2n/3) + O(n), which by the master theorem is $O(n\log n)$, which is slower than linear time.

The median-of-medians algorithm could use a sublist size greater than 5—for example, 7—and maintain a linear running time. However, we need to keep the sublist size as small as we can so that sorting the sublists can be done in what is effectively constant time.