# INDEX

**Program No** : 01
**Program Title** : Complexity Analysis of Various Sorting Algorithm.

## Machine Configuration:

Device name : DESKTOP-PU02UI4
Processor : Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz 1.50 GHz
Installed RAM : 8.00 GB (7.78 GB usable)
System type : 64-bit operating system, x64-based processor
Pen and touch : No pen or touch input is available for this display

## Complexity Analysis:

The time complexity of an algorithm can be measured by the number of operations it completes to reach the solution. The operation can be comparison between two numbers, addition, subtraction, multiplication or division of two numbers or any kind of basic operation.
In the bubble sort algorithm, in a single pass, it compares two adjacent elements and swaps them if needed and this process continues until comparing each element once with it's adjacent element. So, after the ith pass, it is guaranteed that total i element will be in it's permanent place. So, in the next pass, total n-i comparisons is needed to find the position of the next element. So, total number of comparisons will be-

$$(n - 1) + (n - 2) + \cdot \cdot \cdot + 2 + 1 =((n - 1)*n)/2$$

Bubble sort will always take this number of comparisons to make a list of numbers completely sorted whether it be fully sorted at the beginning or at some intermediate state. As the bubble sort uses $n*(n - 1)/2$ comparisons, it has $O(n^2)$ worst-case complexity in terms of the number of comparisons used.

In the insertion sort, ith element is placed into the previously sorted i-1 elements. Here we use a linear search technique to find the place for a particular element as we keep comparing every element with the ith element until we find a element greater than the ith element or we reach the end of the list. Therefore, in the worst case, i comparisons are required to insert the ith element into the correct position. Therefore, the total number of comparisons used by the insertion sort to sort a list of n elements is

$$1+2 + 3 + \cdot \cdot \cdot + n-1 =((n - 1)*n)/2$$

Therefore, we conclude that the insertion sort has worst-case complexity $O(n^2)$.

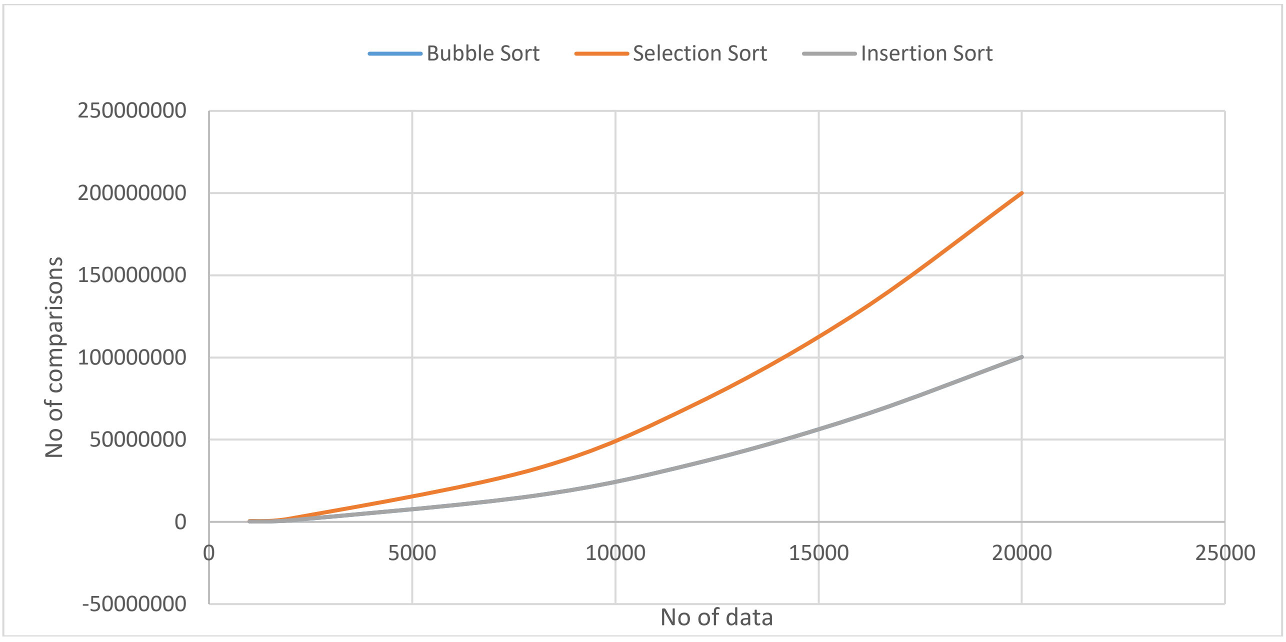
In the selection sort, in the ith pass, we compare n-i elements with i and find the minimum value and if we find any index j that has a minimum value than i, we simply swap the two elements and proceed to the next step. So, total comparisons needed here is as previous $((n - 1)*n)/2$. So, we conclude that the worst case complexity of selection sort is $O(n^2)$.

## Data Table:

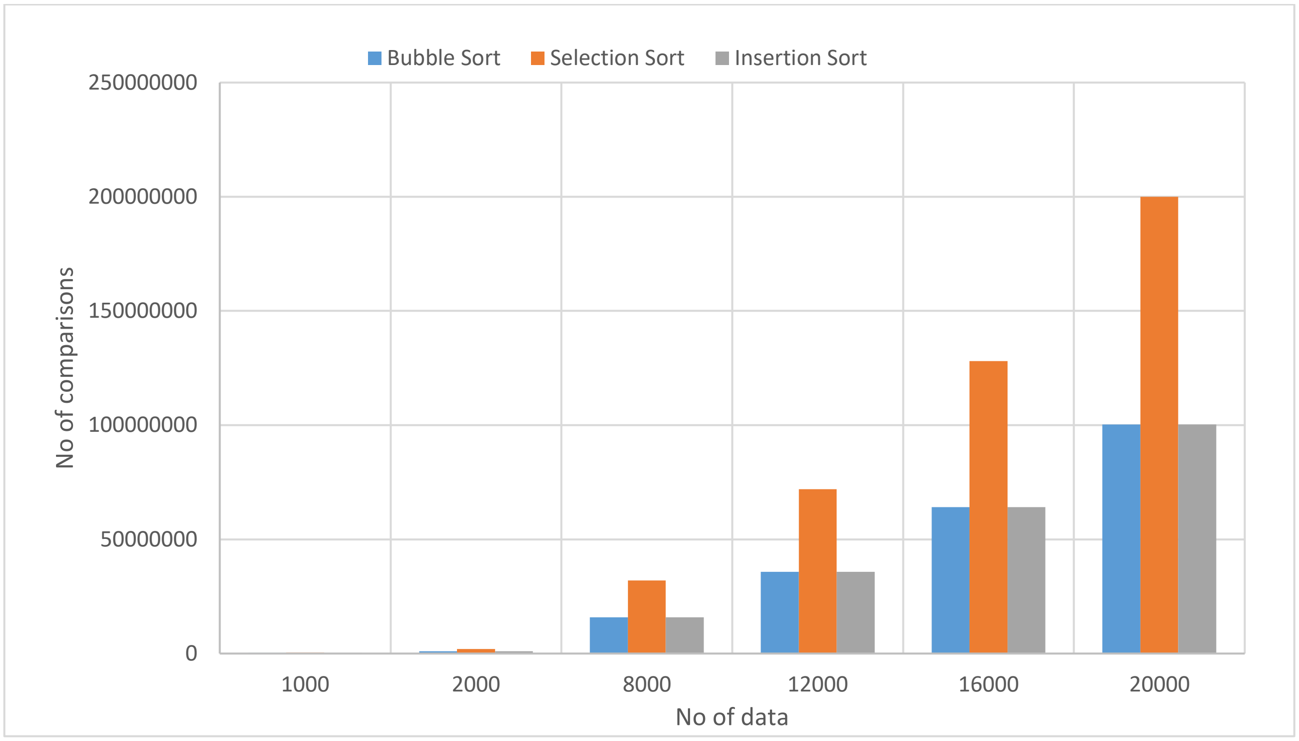
**Table:**

| No. of Data | Bubble Sort | Selection Sort | Insertion Sort |
|---|---|---|---|
| 1000 | 248654 | 499500 | 248654 |
| 2000 | 982178 | 1999000 | 982178 |
| 8000 | 15862980 | 31996000 | 15862980 |
| 12000 | 35735360 | 71994000 | 35735360 |
| 16000 | 64131290 | 127992000 | 64131290 |
| 20000 | 100275325 | 199990000 | 100275325 |

## Graph:



## Bar Graph:



## Table & Graph Analysis:

From the above table, we can say that the required time of bubble sort and the insertion sort is same where the selection sort's is different from other.
In the first graph, we don't see the bubble sort line because bubble sort line is overlapped by insertion sort line as their values are same. But in the bar graph all the three sorts are shown differently in the x axis, so we can see three different graphs.
According the algorithm, the worst complexity is $\theta(n^2)$. But here in the table we can't find any value of worst complexity and the best case complexity is $\theta(n)$ for bubble sort and insertion sort where the selection sort is same as $\theta(n^2)$. So according to the best case and the worst case, the bubble sort and the insertion sort for average case is less than the selection sort where we can see in the table.

**Program No : 02**
**Program Title : Complexity Analysis of Searching Algorithm.**

## A. Linear Search:

### Complexity of Linear Search:

1. The Best Case:
   If the element to be searched is in the first position, then we only need one comparison to find that element. Therefore, Best Case Time Complexity of Linear Search is $O(n^2)$.

2. The worst case:
   If the element to be searched is in the last position or not present in the list then we need to n comparisons to find out the element or be sure that it is not present in the list. Hence, the worst case Time Complexity of Linear Search is $O(n^2)$.

3. Average case:
   Let there be N distinct numbers: $a_1$, $a_2$, ..., $a_{n-1}$, $a_n$. We need to find element P. There are two cases:

   • Case 1: The element P can be in N distinct indexes from 0 to N-1.
   • Case 2: There will be a case when the element P is not present in the list.

   There are 1 in case 1 and N in case 2. So, there are N+1 distinct cases to be considered in total. If element P is in index K, then Linear Search will do K+1 comparisons.
   Number of comparisons for all cases in case 1 =
   Comparisons if element is in index 0 + Comparisons if element is in index 1 + ... + Comparisons if element is in index N-1
   = 1 + 2 + ... + N
   = N * (N+1) / 2 comparisons

   If element P is not in the list, then Linear Search will do N comparisons.
   Number of comparisons for all cases in case 2 = N

   Therefore, total number of comparisons for all N+1 cases = N * (N+1) / 2 + N
   = N * ((N+1)/2 + 1)
   Average number of comparisons = ( N * ((N+1)/2 + 1) ) / (N+1)
   = N/2 + N/(N+1)
   The dominant term in "Average number of comparisons" is N/2. So, the Average Case Time Complexity of Linear Search is O(N).

## Code:

### 1. Iterative Implementation:

```
    #include<bits/stdc++.h>
    #include<ctime>
    using namespace std;

    int main(){
        ifstream input("Input.txt");
        int n, i, c = 0, x, poss = 0;
        input>>n;
        int arr[n];
            for (i = 0; i < n; i++){  // create an array from file with
random    numbers
            input>>arr[i];
        }
        cout<<"Enter the number to be searched: ";
        cin>>x;
        for(i = 0; i < n; i++){
        if(arr[i]==x){
            poss = i+1;
            break;
        }
       else
           poss = 0;
           c++;
        }
        if(poss)
            cout<<x<<" is in the position of "<<poss<<endl;
        else
            cout<<x<<" is not found in the list."<<endl;
        cout<<"The time complexity is "<<c<<endl;
         clock_t end = clock();
        double time_taken = end/CLOCKS_PER_SEC;
        time_taken = time_taken*1000;
        printf("Linear search(iterative) took %f milliseconds to execute \n",
time_taken);
        return 0;    }
```

## 2. Recursive Implementation:

```
#include<bits/stdc++.h>
#include<time.h>
using namespace std;

int searchElement(int arr[], int s, int x) {
    s--;
    if (s < 0) {
        return -1;
    }
    if (arr[s] == x) {
        return s;
    }
    return searchElement(arr, s, x);
}

int main(){
    ifstream input("Input.txt");
    int n, i, x;
    input>>n;
    int arr[n];

    for (i = 0; i < n; i++){ // create an array from file with random numbers

        input>>arr[i];
    }
    cout<<"Enter the number to be searched : ";
    cin>>x;
    int l = searchElement(arr, n, x);
    if (l != -1)
        cout << x<<"is in the list!"<<endl;
    else
        cout << x<<"is not in the list!"<<endl;
    clock_t stop = clock();
    double time_taken = stop/CLOCKS_PER_SEC;
    time_taken = time_taken*1000;
    printf("Linear search(recursive) took %f milliseconds to execute \n",
time_taken);

    return 0;
}
```

## Runtime of Linear Search:

| | Required time (microseconds) | |
|---|---|---|
| **No. of Data** | **Iterative** | **Recursive** |
| 1000 | 4000 | 3000 |
| 5000 | 8000 | 4000 |
| 10000 | 16000 | 14000 |
| 15000 | 14000 | 5000 |

## B. Binary Search:

### Complexity of Binary Search:

1. **The Best Case:** The element to be search is in the middle of the list. In this case, the element is found in the first step itself and this involves 1 comparison. Therefore, Best Case Time Complexity of Binary Search is O(1).
2. **The worst Case:** The element is to search is in the first index or last index. In this case, the total number of comparisons required is log N comparisons. Therefore, Worst Case Time Complexity of Binary Search is O(log N).
3. **The Average Case:** Let there be N distinct numbers: $a_1$, $a_2$, ..., $a_{n-1}$, $a_n$. We need to find element P. There are two cases:
   Case 1: The element P can be in N distinct indexes from 0 to N-1.
   Case 2: There will be a case when the element P is not present in the list.
   There are N case 1 and 1 case 2. So, there are N+1 distinct cases to consider in total. If element P is in index K, then Binary Search will do K+1 comparisons. This is because: The element at index N/2 can be found in 1 comparison as Binary Search starts from middle. Similarly, in the 2nd comparisons, elements at index N/4 and 3N/4 are compared based on the result of 1st comparison. On this line, in the 3rd comparison, elements at index N/8, 3N/8, 5N/8, 7N/8 are compared based on the result of 2nd comparison.
   Based on this, we know that:
   Elements requiring 1 comparison: 1
   Elements requiring 2 comparisons: 2
   Elements requiring 3 comparisons: 4
   Therefore, Elements requiring I comparisons: $2^{l-1}$

   The maximum number of comparisons = Number of times N is divided by 2 so that result is 1 = Comparisons to reach 1st element = logN comparisons
   I can vary from 0 to logN.

Total number of comparisons = 1 * (Elements requiring 1 comparison) + 2 * (Elements requiring 2 comparisons) + ... + logN * (Elements requiring logN comparisons)
Total number of comparisons = 1 * (1) + 2 * (2) + 3 * (4) + ... + logN * (2^(logN-1))
Total number of comparisons = 1 + 4 + 12 + 32 + ... = 2^logN * (logN - 1) + 1
Total number of comparisons = N * (logN - 1) + 1
Total number of cases = N+1

Therefore, average number of comparisons = ( N * (logN - 1) + 1 ) / (N+1)
Average number of comparisons = N * logN / (N+1) - N/(N+1) + 1/(N+1)
The dominant term is N * logN / (N+1) which is approximately logN. Therefore, Average Case Time Complexity of Binary Search is O(logN).

## Code:

### 1. Iterative Implementation:

```
#include <bits/stdc++.h>
#include<ctime>
using namespace std;

int main()
{
  int n,i,j,m,loc=0,x,c=0;
    cout<<"Enter the value to be searched : ";
  cin>>x;
  clock_t start= clock();
  ifstream input("input.txt");

  input>>n;
  int arr[n+1];

  for (i = 0; i < n; i++)  // create an array from file with random numbers
  {
      input>>arr[i];
  }
  i=0, j=n, loc=0;


  while(i<=j){
          c++;
  m=(i+j)/2;
  if(arr[m]==x)
    {loc=m;break;}
  else if(arr[m]<x)
    i=m+1;
  else
    j=m-1;}

  if(loc)cout<<x<<" is in the array at the position of "<<loc<<endl;
  else cout<<x<<" is not found in the array"<<endl;
  cout<<"The time complexity is "<<c<<endl;
  clock_t stop = clock();
  double time_taken = (double)(start-stop)/CLOCKS_PER_SEC;
  time_taken = time_taken*1000000;
  printf("Binary search(iterative) took %f microseconds to execute \n",
time_taken);
  return 0;
}
```

## 2. Recursive Implementation:

```
  #include<bits/stdc++.h>
  #include<time.h>
  using namespace std;
  int binary(int arr[],int l,int r,int x)
  {
  if (r >= l) {
      int mid = l + (r - l) / 2;
      if (arr[mid] == x)
          return mid;
      if (arr[mid] > x)
          return binary(arr, l, mid - 1, x);
      return binary(arr, mid + 1, r, x);
  }
  return -1;
  }
  int main()
  {
  int n,i,j,m,loc=0,x,c=0;
  cout<<"Enter the value to be searched : ";
  cin>>x;
  clock_t start = clock();
  ifstream input("input.txt");

  input>>n;
  int arr[n+1];

  for (i = 0; i < n; i++) // create an array from file with random numbers
  {
      input>>arr[i];
  }
```

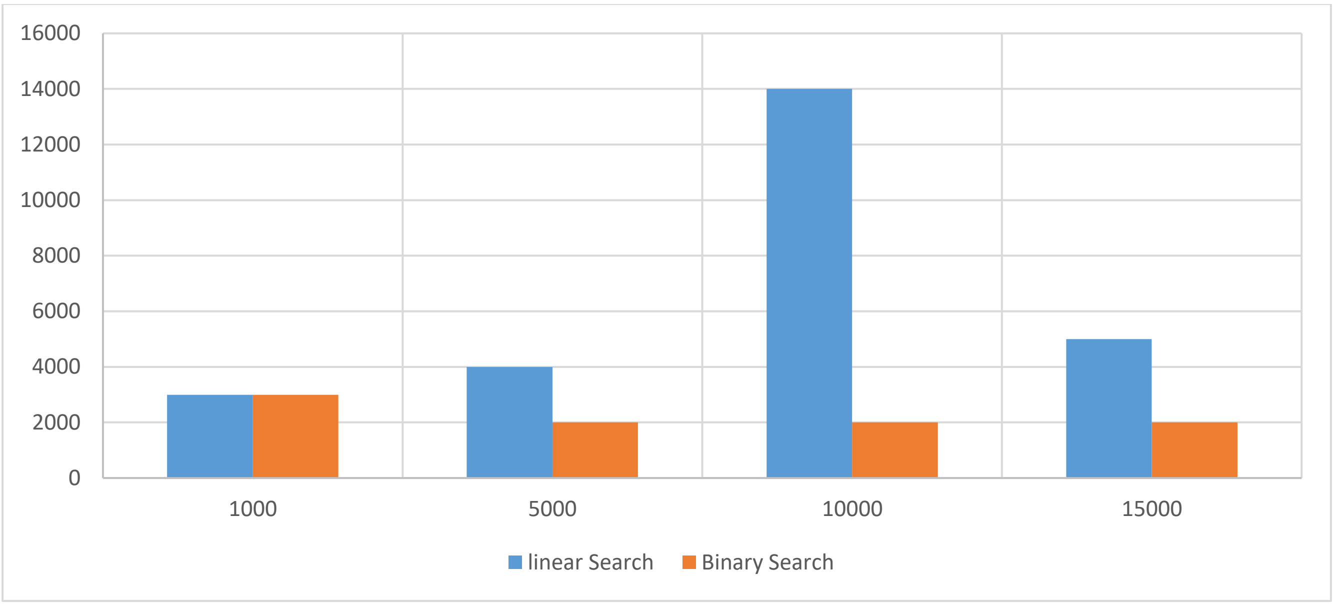
```
i=0, j=n, loc=0;


int t = binary(arr,0,n-1, x);
if(t==-1)
    cout<<"Not founded!"<<endl;
else
    cout<<"Founded!"<<endl;
clock_t stop = clock();

double time_taken = (double)(start-stop)/CLOCKS_PER_SEC;
time_taken = time_taken*1000000;
printf("Binary search(recursive) took %f microseconds to execute \n",
time_taken);
return 0;
}
```

## Time required for Binary Search:

| No. of Data | Iterative | Recursive |
|---|---|---|
| 1000 | 3000 | 3000 |
| 5000 | 4000 | 2000 |
| 10000 | 5000 | 2000 |
| 15000 | 7000 | 2000 |

## Comparison of Linear and Binary Search:

If we compare the time taken by Linear and Binary search then we get the data which is shown in the below graph-



From the graph we see that linear search takes much more time compared to binary search as in the linear search we have to go through every element one by one and check if the element is present in the list or not. And if the element is not present in the list, then first we have to traverse the whole list and then only we can say that the element is not present in the list. But in binary search, we directly divide the data into two parts and search for only one part. Therefore it takes much less time. But if the data we are looking for is somewhere at the beginning of the list, then linear search will find it faster compared to binary search. That's why we were considering 1000 data, we see that time taken by both linear and binary search is quite close.

# INDEX

**Program No** : 01

**Program Title** : Complexity Analysis of Finding Maximum and Minimum Value [Max-Min Algorithm]

## Machine Configuration:

| | |
|---|---|
| Device name | : DESKTOP-PU02UI4 |
| Processor | : Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz 1.50 GHz |
| Installed RAM | : 8.00 GB (7.78 GB usable) |
| System type | : 64-bit operating system, x64-based processor |
| Pen and touch | : No pen or touch input is available for this display |

## Complexity Analysis:

### Brute-force Method:

Brute-force method or Straightforward method requires 2(n-1) element comparisons in the best, average and worst case. In every case, there is total n number of comparisons and the loop continues from 1 to n-1. SO, total comparison becomes 2(n-1)

### Divide & Conquer Method:

If *T(n)* represents this number, then the resulting recurrence relation is

$$T(n) = 0, \quad \text{if } n=1$$
$$T(n) = 1, \quad \text{if } n=2$$
$$T(n) = 2T(n/2) + T(n/2) + 2 \quad \text{if } n > 2, \text{ c is a constant}$$

When n is a power of 2, $n = 2^k$, we can solve this equation by successive substitutions

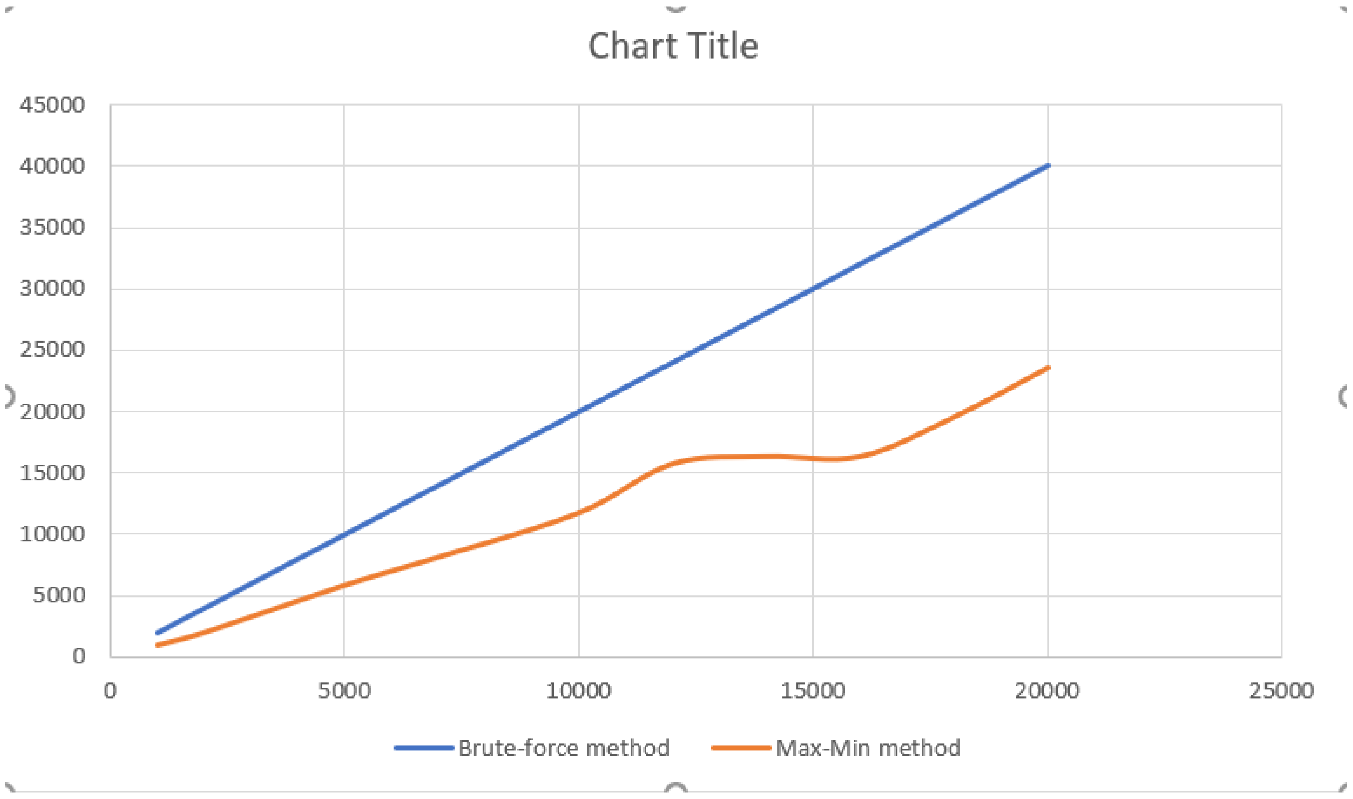
$$\begin{aligned} T(n) &= 2T(n/2) + 2 \\ &= 2(2T(n/4) + 2) + 2 \\ &= 4T(n/8) + 4 + 2 \\ &\;\;. \\ &\;\;. \\ &= 2^{k-1}T(2) + \sum_{1\le i\le k+1}(2^i) \\ &= 2^{k-1} + 2^k - 2 \\ &= 3n/2 - 2 \end{aligned}$$

Note that 3n/2 – 2 is the best, average and worst case number comparisons when n is a power of 2.

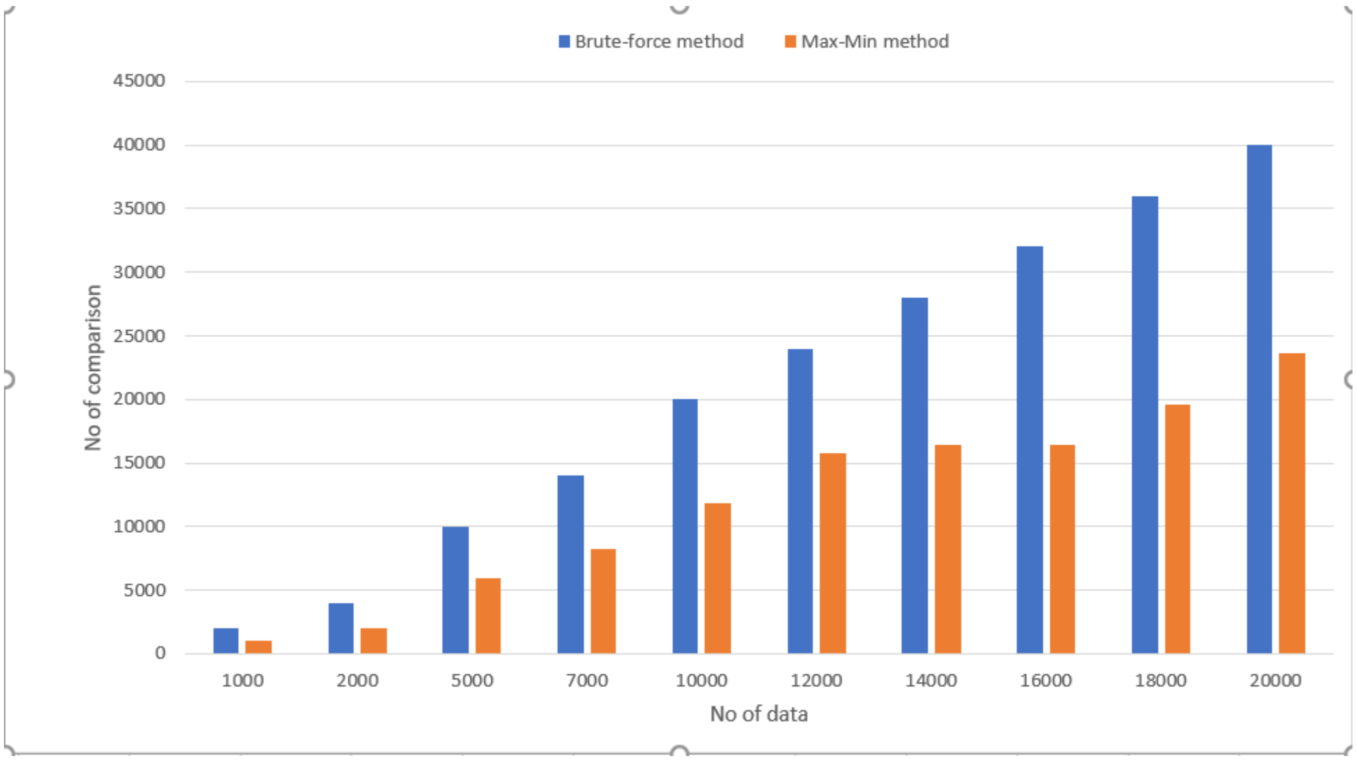
## Time required to find Maximum and Minimum:

| | Finding Maximum and Minimum | |
|---|---|---|
| **No. of Data** | **Brute-force method** | **Max/Min method** |
| 1000 | 1998 | 1023 |
| 2000 | 3998 | 2047 |
| 5000 | 9998 | 5903 |
| 7000 | 13998 | 8191 |
| 10000 | 19998 | 11807 |
| 12000 | 23998 | 15807 |
| 14000 | 27998 | 16383 |
| 16000 | 31998 | 16383 |
| 18000 | 35998 | 19615 |
| 20000 | 39998 | 23615 |

## Graph:



## Bar Graph:



## Table & Graph Analysis:

From the table and graph, we can see that the value of Brute-force method is increasing gradually but the value of divide & conquer method is increasing but not gradually. The Brute-force method graph is linear but the divide & conquer method not. According the complexity analysis, the Brute-force method is 2(n - 1) where divide & conquer method is 3n/2 – 2 for the worst case, average case and best case. But in divide and conquer method when the n is 14000 and 16000, the complexity shows the same value which is 16383 for any input.

From the above table and graphs, we can see that the recursive max min algorithm is faster than the straight forward method. Both the curves follow linearity. And we can conclude that both the methods have complexity O(n). But in practice Divide and conquer method is comparatively faster than the straight forward method.

**Program No** : 02
**Program Title** : Complexity Analysis of Sorting Algorithm.

**Machine Configuration:**

| | |
|---|---|
| Device name | : DESKTOP-PU02UI4 |
| Processor | : Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz 1.50 GHz |
| Installed RAM | : 8.00 GB (7.78 GB usable) |
| System type | : 64-bit operating system, x64-based processor |
| Pen and touch | : No pen or touch input is available for this display |

**Complexity Analysis:**

Merge Sort:

If the time for the merging operation is proportional to n, then the computing time for merge sort can be described by the recurrence relation

$$T(n) = a, \qquad \text{if } n=1 \text{ \& } a \text{ is a constant}$$
$$T(n) = 2T(n/2) + cn \qquad \text{if } n > 1, c \text{ is a constant}$$

When n is a power of 2, $n = 2^k$, we can solve this equation by successive substitutions

$$\begin{aligned} T(n) &= 2(2T(n/4) + cn/2) + cn \\ &= 4T(n/4) + 2cn \\ &= 4(2T(n/8) + cn/4) + 2cn \\ &. \\ &. \\ &= 2^kT(1) + kcn \\ &= an + cn \log n \end{aligned}$$

It is easy to see that if $2^k < n \le 2^{k+1}$, then $T(n) \le T(2^{k+1})$. Therefore

$$T(n) = O(n \log n)$$

Quick Sort:

In general idea we divide the array into two subarrays so that the sorted subarrays. This is accomplished by rearranging the elements in subarrays to a particular point. The point is called pivot. The elements of one side of the pivot are less than then element at pivot and the other half the elements greater than the element at pivot. Thus, the pivot is fixed position of the array. We then divide the two halves into two subarrays and continue the process.

In analyzing Quicksort, we count only the number of element comparisons *C(n)*. It is easy to see that the frequency count of other operations is of the same order as *C(n)*. We make the following assumptions: the n elements to be sort are distinct and the input distribution is such that the partition element v = a[m] in the call to Partition (a, m, p) has an equal probability of being the *i* th smallest element, $1 \le i \le p-m$, *in a[m:p-1]*.

The worst-case value $C_w(n)$ of $C(n)$. In each call the element comparisons is at most p - m – 1. Let r be the total number of elements in all the calls to Partition at any level of recursion. At level one only one call, Partition(a, 1,n+1) is made and r = n. One side complexity is O(n) and the other side is O(n). The total complexity is $O(n^2)$.

For the average case:

$C_A(n) = n + 1 + 1/n \sum_{1\le k\le n}[C_A(k-1) + C_A(n-k)]$

$nC_A(n) = n(n + 1) + 2[C_A(0) + C_A(1) + \dots\dots\dots + C_A(n-1)]$ ---------------(i)

$(n-1)C_A(n-1) = n(n - 1) + 2[C_A(0) + C_A(1) + \dots\dots\dots + C_A(n-2)]$-----------(ii)

Subtracting from (ii) to (i), we get-

$nC_A(n) - (n-1)C_A(n-1) = 2n + 2C_A(n-1)$

$C_A(n)/(n+1) = C_A(n-1)/n+2/(n+1)$

$$\frac{C(n)}{n+1} = \frac{C(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1}$$

$$= \frac{C(n-3)}{n-2} + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1}$$

.

.

$$= \frac{C(1)}{2} + 2\sum_{3\le k\le n+1}\left(\frac{1}{k}\right)$$

$$= 2\sum_{3\le k\le n+1}\left(\frac{1}{k}\right)$$

$C_A(n) \le 2(n+1)[\log_e(n+2) - \log_e 2] = O(n \log n)$.

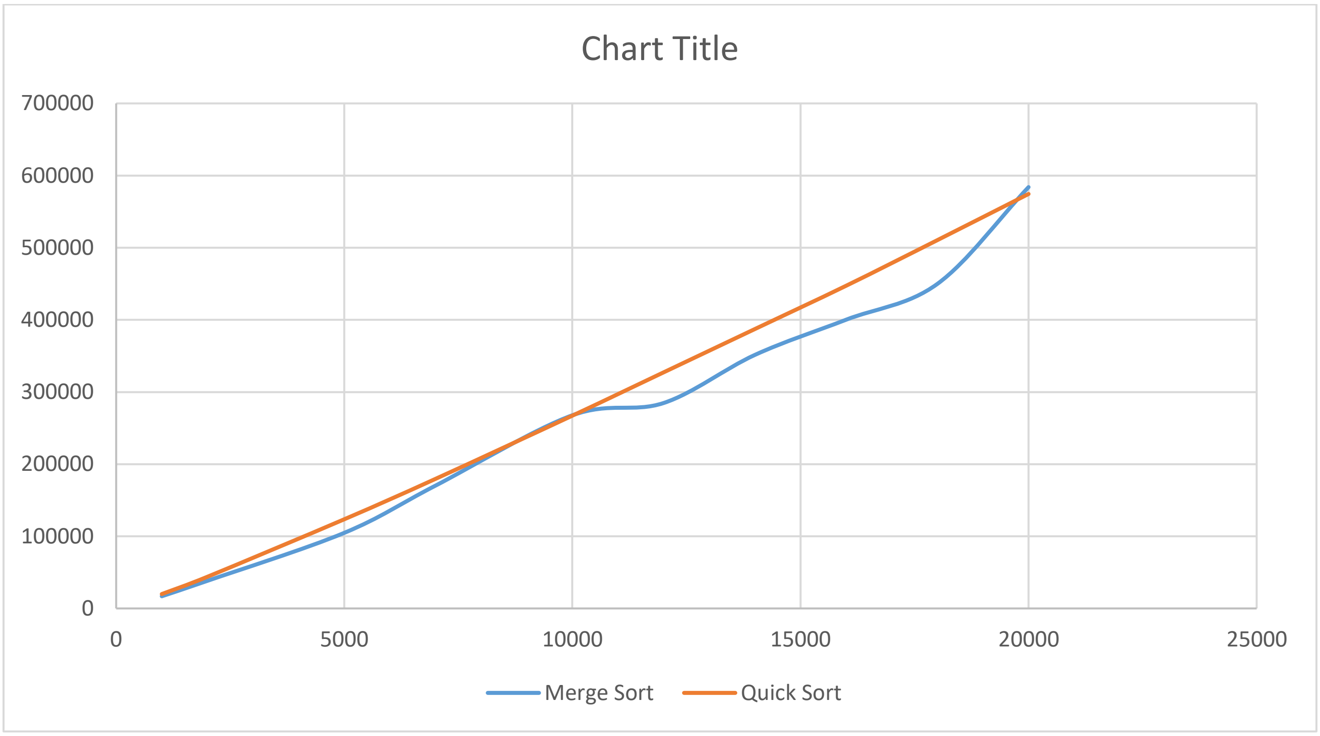
Thus, T(n) = O(n log n) for best and average case. $O(n^2)$ for worst case.

## Data Table:

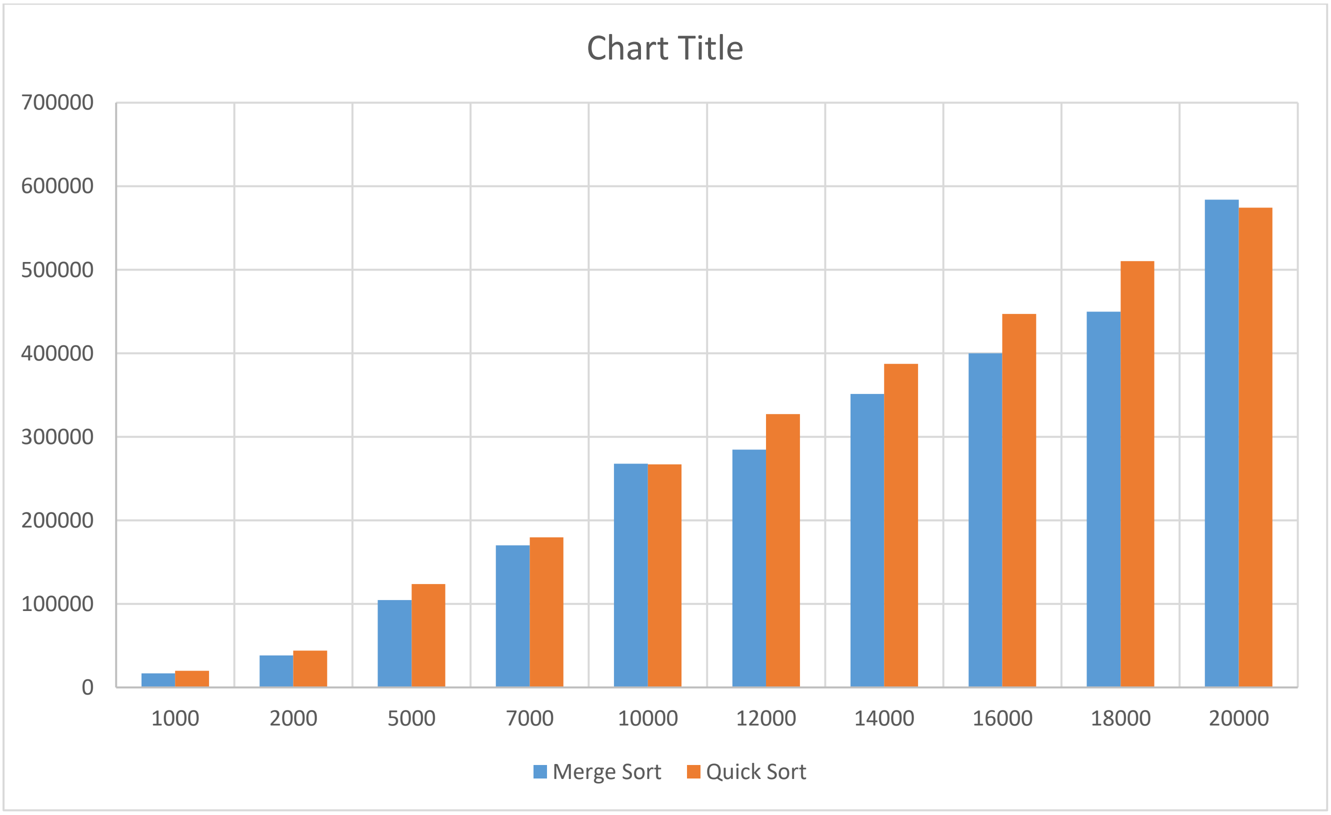
**Table:**

| No. of Data | Merge Sort | Quick Sort |
|---|---|---|
| 1000 | 16912 | 19952 |
| 2000 | 38309 | 43904 |
| 5000 | 104763 | 123616 |
| 7000 | 170258 | 179616 |
| 10000 | 267770 | 267232 |
| 12000 | 284745 | 327232 |
| 14000 | 351384 | 387232 |
| 16000 | 400189 | 447232 |
| 18000 | 449802 | 510464 |
| 20000 | 583944 | 574464 |

## Graph:



## Bar Graph:



## Table & Graph Analysis:

From the table and graph, we can see that. the merge sort graph is linear but the quick sort is not. According the complexity analysis, the merge sort is an + cn log n, where log n value differs when the difference value is so high. But cn adjust the value. If n = 1, then the complexity is a which is also constant. But the average case of complexity of quick sort is $2(n+1)[\log_e(n+2) – \log_e 2] = O(n \log n)$ which value differ from the merge sort. So some time quick sort is increasing gradually but not all the time and the worst case is $O(n^2)$ which is nonlinear.

From the above table and graphs, it is noticed that for small values of n quick sort
is faster than the merge sort. But for big number of data merge sort is faster. This is because for worst case quick sort has the complexity of $O(n^2)$. Whereas merge sort has always the complexity of O(n log n).