Name	Harsh Chandra
UID no.	2021700013
Experiment No.	1

AIM:

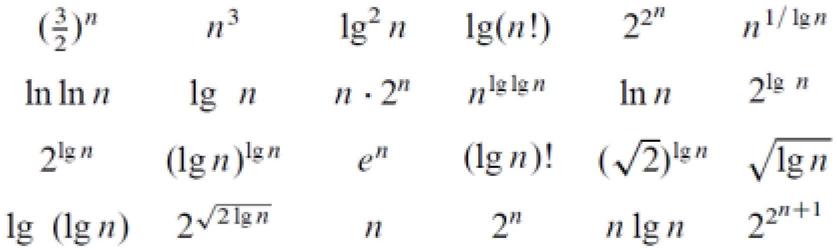
1-a – To implement the various functions e.g. linear, non-linear, quadratic, exponential etc.

1-b – Experiment on finding the running time of an algorithm.

Program 1

PROBLEM STATEMENT :

For this experiment, you have to implement at least 10 functions from the following list.



$(\frac{3}{2})^n$	n^3	$\lg^2 n$	$\lg(n!)$	2^{2^n}	$n^{1/\lg n}$
$\ln \ln n$	$\lg\ \ n$	$n \cdot 2^n$	$n^{\lg \lg n}$	$\ln n$	$2^{\lg\ n}$
$2^{\lg n}$	$(\lg n)^{\lg n}$	e^n	$(\lg n)!$	$(\sqrt{2})^{\lg n}$	$\sqrt{\lg n}$
$\lg\ \ (\lg n)$	$2^{\sqrt{2 \lg n}}$	n	2^n	$n \lg n$	$2^{2^{n+1}}$

Note – lg denotes for log2 and le denotes loge

The input (i.e. n) to all the above functions varies from 0 to 100 with increment of 1. Then add the function n! in the
list and execute the same for n from 0 to 20.

ALGORITHM/ THEORY:

In this program we have defined 10 functions and call them for numbers ranging from 0 to 99.

Functions used:
n*logn logn log(logn) √log2n 2^logn 2^log2n log2n
√2^(log2n) 2^√2*log2n n

Time Complexity: The time complexity of an algorithm **quantifies the amount of time taken by an algorithm to run as a function of the length**

of the input. Note that the time to run is a function of the length of the input and not the actual execution time of the machine on which the algorithm is running on.

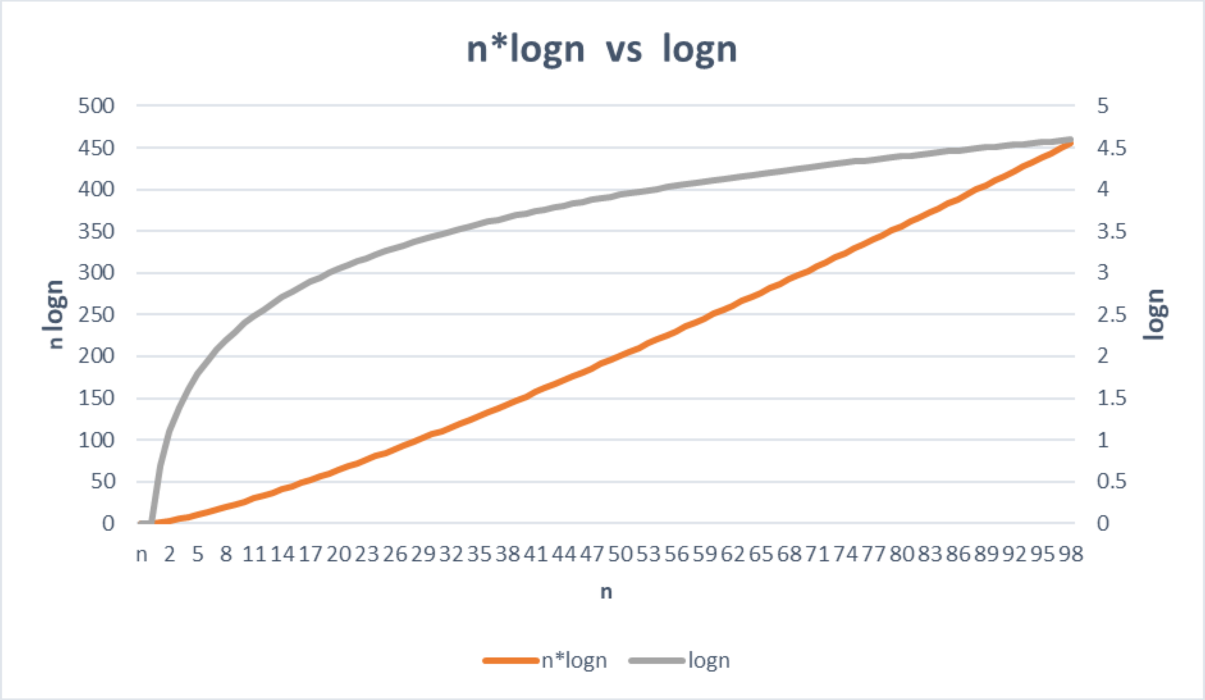
Order of growth is how the time of execution depends on the length of the input. In the above example, it is clearly evident that the time of execution quadratically depends on the length of the array. Order of growth will help to compute the running time with ease

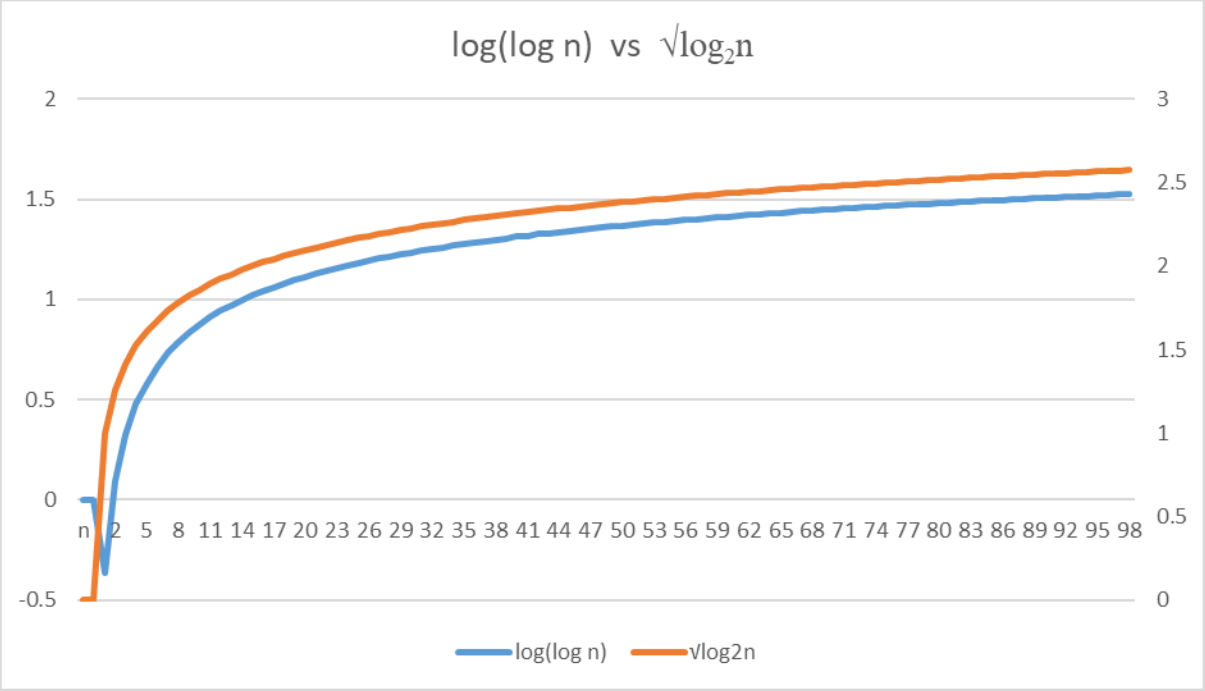
PROGRAM:

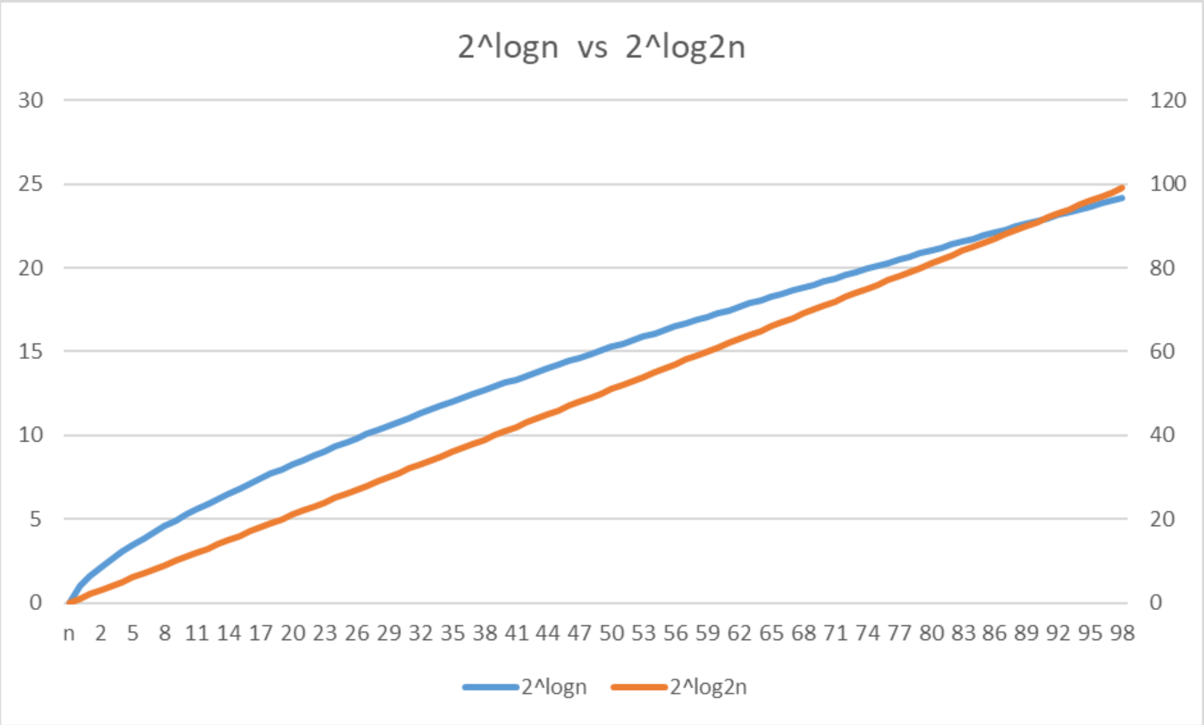
```
#include <stdio.h>
#include <math.h>

double fun1(int n){        // n log(n)
    double ans=n*log(n);
    return ans;
}
double fun2(int n){         // logn
    double ans=log(n);
    return ans;
}
double fun3(int n){         //  log(log n)
    double ans=log((log)(n));
    return ans;
}
double fun4(int n){         //  sqrt(log2 n)
    double ans=sqrt((log2)(n));
    return ans;
}
double fun5(int n){         //  2^log n
    double ans=pow(2,log(n));
    return ans;
}
double fun6(int n){         //  2^log2 n
    double ans=pow(2,log2(n));
    return ans;
}
double fun7(int n){         //  log2 n
    double ans=log2(n);
    return ans;
```

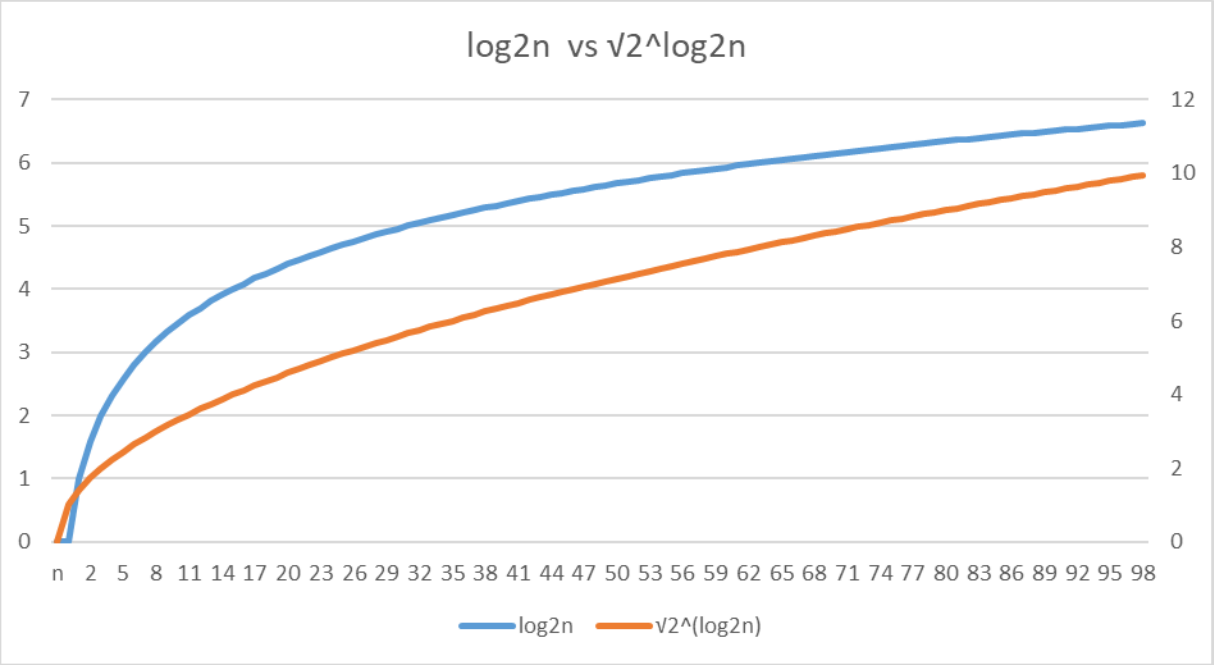
```
}
double fun8(int n){         //  root(2)^log2n
    double ans=pow(1.4142,log2(n));
    return ans;
}
double fun9(int n){          //  2^root(2*log2n)
    double root=sqrt(2*log2(n));
    double ans=pow(2,root);
    return ans;
}
double fun10(int n){          // n
    return n;
}

long long factorial(int n){
    long long pro=1;
    for(int i=1;i<=n;i++){
        pro*=i;
    }
    printf("%lld\n",pro);
}

int main()
{
    int n=100;
    printf("n\t");
    printf("Fun1\t");
    printf("Fun2\t");
    printf("Fun3\t");
    printf("Fun4\t");
    printf("Fun5\t");
    printf("Fun6\t");
    printf("Fun7\t");
    printf("Fun8\t");
    printf("Fun9\t");
    printf("Fun10\n");

    for(int i=0;i<n;i++){
        printf("%d\t",i);
        double f1=fun1(i);
        printf("%.3f\t",f1);
        double f2=fun2(i);
        printf("%.3f\t",f2);
        double f3=fun3(i);
        printf("%.3f\t",f3);
        double f4=fun4(i);
        printf("%.3f\t",f4);
        double f5=fun5(i);
        printf("%.3f\t",f5);
        double f6=fun6(i);
        printf("%.3f\t",f6);
        double f7=fun7(i);
        printf("%.3f\t",f7);
        double f8=fun8(i);
        printf("%.3f\t",f8);
        double f9=fun9(i);
        printf("%.3f\t",f9);
        double f10=fun10(i);
        printf("%.3f\t",f10);

        printf("\n");
    }
    printf("factorials:\n");
    for(int i=1;i<=20;i++){
        factorial(i);
    }

    return 0;
}
```

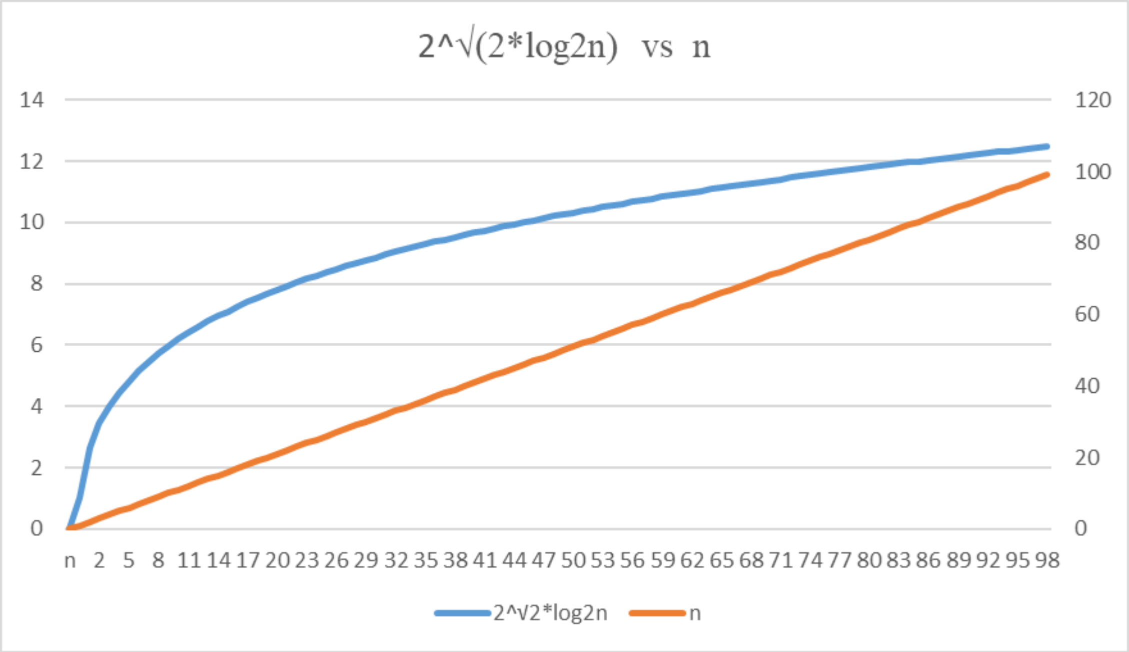
RESULT:

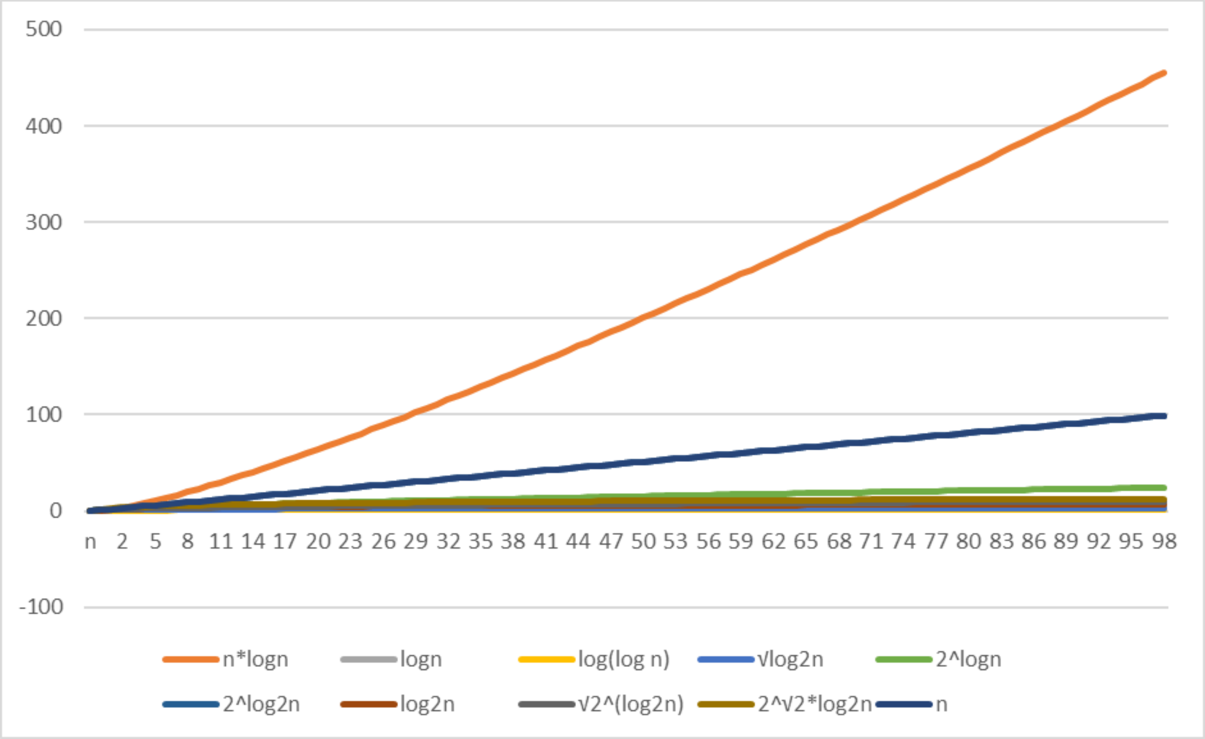

n*logn vs logn
500
450
400
350
300
250
200
150
100
50
0
n logn
5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
logn
n 2 5 8 11 14 17 20 23 26 29 32 35 38 41 44 47 50 53 56 59 62 65 68 71 74 77 80 83 86 89 92 95 98
n
n*logn
logn


log(log n) vs √log$_2$n
2
1.5
1
0.5
0
-0.5
3
2.5
2
1.5
1
0.5
0
n 2 5 8 11 14 17 20 23 26 29 32 35 38 41 44 47 50 53 56 59 62 65 68 71 74 77 80 83 86 89 92 95 98
log(log n)
√log2n


2^logn vs 2^log2n
30
25
20
15
10
5
0
120
100
80
60
40
20
0
n 2 5 8 11 14 17 20 23 26 29 32 35 38 41 44 47 50 53 56 59 62 65 68 71 74 77 80 83 86 89 92 95 98
2^logn
2^log2n


log2n vs √2^log2n
7
6
5
4
3
2
1
0
12
10
8
6
4
2
0
n 2 5 8 11 14 17 20 23 26 29 32 35 38 41 44 47 50 53 56 59 62 65 68 71 74 77 80 83 86 89 92 95 98
log2n
√2^(log2n)





CONCLUSION:	Implemented various linear, non-linear, quadratic, exponential functions and observed the difference in result for all values of n from 0 to 100.

	From the chosen functions function n*logn grows rapidly as the input number increases function n (linear) increases linearly with the input and is the second fastest growing function among others. The graph representation helps us to identify and compare various functions.
Program 2	
PROBLEM STATEMENT :	For this experiment, you need to implement two sorting algorithms namely Insertion and Selection sort methods. Compare these algorithms based on time and space complexity. Time required to sorting algorithms can be performed using high_resolution_clock::now() under namespace std::chrono. You have togenerate1,00,000 integer numbers using C/C++ Rand function and save them in a text file. Both the sorting algorithms uses these 1,00,000 integer numbers as input as follows. Each sorting algorithm sorts a block of 100 integers numbers with array indexes numbers A[0..99], A[0..199], A[0..299],..., A[0..99999]. You need to use high_resolution_clock::now() function to find the time required for 100, 200, 300.... 100000 integer numbers. Finally, compare two algorithms namely Insertion and Selection by plotting the time required to sort 100000 integers using LibreOffice Calc/MS Excel. The x-axis of 2-D plot represents the block no. of 1000 blocks. The y-axis of 2-D plot representsthe tunning time to sort 1000 blocks of 100,200,300,...,100000 integer numbers.
ALGORITHM/ THEORY:	Selection Sort: In selection sort, the smallest value among the unsorted elements of the array is selected in every pass and inserted to its appropriate position into the array. It is also the simplest algorithm. It is an in-place comparison sorting algorithm. In this algorithm, the array is divided into two parts, first is sorted part, and another one is the unsorted part. Initially, the sorted part of the array is empty, and unsorted part is the given array. Sorted part is placed at the left, while the unsorted part is placed at the right. The average and worst-case complexity of selection sort is **O(n^2)**, where **n** is the number of items. Due to this, it is not suitable for large data sets.

Step 1 − Set MIN to location ith index.
Step 2 − Search the minimum element in the list
Step 3 − Swap with value at location MIN
Step 4 − Increment MIN to point to next element
Step 5 − Repeat until list is sorted

Insertion Sort:

The idea behind the insertion sort is that first take one element, iterate it through the sorted array. Although it is simple to use, it is not appropriate for large data sets as the time complexity of insertion sort in the average case and worst case is **O(n^2)**, where n is the number of items. Insertion sort is less efficient than the other sorting algorithms like heap sort, quick sort, merge sort, etc.

Step 1 - If the element is the first element, assume that it is already sorted.
Step2 - Pick the next element, and store it separately in a **key.**
Step3 - Now, compare the **key** with all elements in the sorted array.
Step 4 - If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.
Step 5 - Insert the value.
Step 6 - Repeat until the array is sorted.

Time Complexity of selection sort is always n*(n - 1)/2, whereas insertion sort has better time complexity as its worst case complexity is n*(n - 1)/2. Generally it will take lesser or equal comparisons then n*(n - 1)/2.

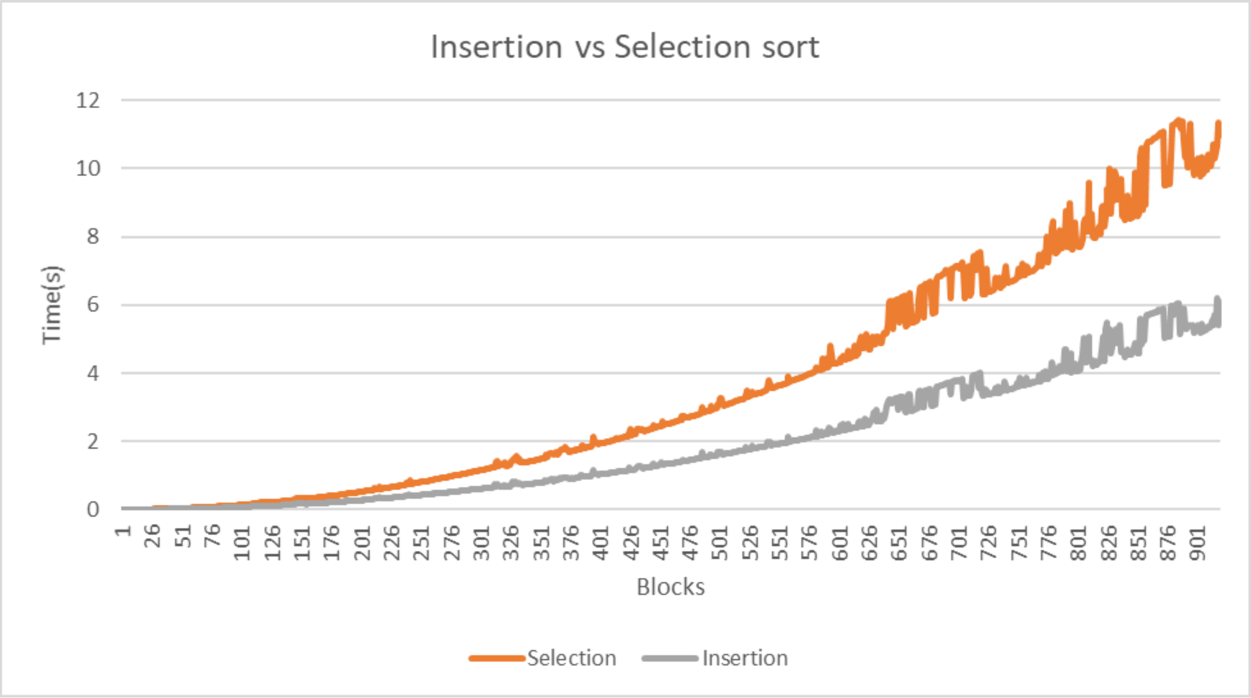
PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void selectionsort(int array[], int end){
  for (int i = 0; i < end; i++)
  {
    int mini = i;
    for (int j = i + 1; j < end; j++)
    {
      if (array[j] < array[mini])
```

```
   {
    mini = j;
   }
  }
  int temp = array[i];
  array[i] = array[mini];
  array[mini] = temp;
 }
}

void insertionsort(int a[], int n){
 for (int i = 1; i < n; i++){
  int key = a[i];
  int j = i - 1;
  while (j >= 0 && a[j] > key){
   a[j + 1] = a[j];
   j = j - 1;
  }
  a[j + 1] = key;
 }
}

int main(){

 FILE *fptr;
 fptr = fopen("randomm.txt", "w");
 if (fptr == NULL){
  printf("ERROR Creating File!");
  exit(1);
 }

 int n = 100000;
 srand(time(0));
 for (int i = 1; i <= n; i++){
  int r = rand() % 100;
  fprintf(fptr, "%d\n", r);
 }
 fclose(fptr);

 int block = 1;
```

```
  printf("Block\tSelection\tInsertion\n");
  for (int i = 100; i <= n; i += 100){
    fptr = fopen("randomm.txt", "r");
    int arr[i];

    for (int j = 0; j < i; j++){
      fscanf(fptr, "%d", &arr[j]);
    }
    clock_t t;
    t = clock();
    selectionsort(arr, i);
    t = clock() - t;
    double time_takenss = ((double)t) / CLOCKS_PER_SEC;
    fclose(fptr);

    fptr = fopen("randomm.txt", "r");
    int arr2[i];
    for (int j = 0; j < i; j++){
      fscanf(fptr, "%d", &arr2[j]);
    }
    clock_t t2;
    t2 = clock();
    insertionsort(arr2, i);
    t2 = clock() - t2;
    double time_takenis = ((double)t2) / CLOCKS_PER_SEC;

    printf("%d\t%f\t%f\n", block, time_takenss, time_takenis);

    fclose(fptr);
    block++;

  }
  return 0;
}
```

RESULT:



Insertion sort takes less time as compared to selection sort.
Both insertion sort and selection sort have an outer loop (over every index), and an inner loop (over a subset of indices). Each pass of the inner loop expands the sorted region by one element, at the expense of the unsorted region, until it runs out of unsorted elements.
The difference is in what the inner loop does:

- In selection sort, the inner loop is over the *unsorted* elements. Each pass selects one element, and moves it to its final location (at the current end of the sorted region).
- In insertion sort, each pass of the inner loop iterates over the *sorted* elements. Sorted elements are displaced until the loop finds the correct place to insert the next unsorted element.

So, in a selection sort, sorted elements are found in output order, and stay put once they are found. Conversely, in an insertion sort, the *unsorted* elements stay put until consumed in input order, while elements of the sorted region keep getting moved around.

CONCLUSION:

Among both of the sorting algorithm, the insertion sort is fast, efficient, stable while selection sort only works efficiently when the small set of elements is involved or the list is partially previously sorted. The number of comparisons made by selection sort is greater than the movements performed whereas in insertion sort the number of times an element is moved or swapped is greater than the comparisons made.

	Implemented various linear, non-linear, quadratic, exponential functions and observed the difference in result for all values of n from 0 to 100.